MOSSTAT

An Interactive Static Rule Checker for MOS VLSI Designs

Timothy E. Johnson
B.S., Lewis and Clark College, 1974

A thesis submitted to the faculty of the
Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering
June, 1986

The thesis "An Interactive Static Rule Checker for MOS VLSI Designs" by Timothy E. Johnson has been examined and approved by the following Examination Committee:

Alan C.(Kit) Bradley, Thesis Research Advisor
Adjunct Assistant Professor,
Department of Computer Science and Engineering

Richard B. Kieburtz
Professor and Chairman,
Department of Computer Science and Engineering

Daniel Hammerstrom
Associate Professor,
Department of Computer Science and Engineering

Shreekant S. Thakkar
Assistant Professor,
Department of Computer Science and Engineering

Abstract

MOSSTAT
An Interactive Static Rule Checker for MOS VLSI Designs

Timothy Johnson, M. S.
Oregon Graduate Center, 1985

Supervising Professor: Kit Bradley

A Static Rule Checker for NMOS and CMOS VLSI Circuits is described. MOSSTAT makes a number of different static rule checks on a circuit. These checks help the user to detect and isolate errors such as improper network connectivity or invalid transistor sizes, and can be run interactively to allow for orderly execution each rule check. The results are stored in a data base. MOSSTAT provides a simple *query* language that allows the user to selectively retrieve this information from the data base. Transistors are classified according to their type and function. Logic gates are also classified according to their style. The results of these analyses are useful in isolating possible circuit design problems.

Acknowledgements

I would first like to express my gratitude to my wife Debbie who managed to stand by me through my 'thesis years' and my son Tyler who has learned to get along without me on the many weekends that were spent away from home.

I would also like to express my thanks to Kit Bradley who wore three hats: my manager at Tektronix, my thesis advisor at OGC and friend. As my manager he proposed a thesis topic of interest to Tektronix, OGC, and myself. As my thesis advisor he gave me considerable guidance in both the project and the thesis and made necessary adjustments when needed to ensure a timely completion of the project. As my friend he gave me much needed encouragement.

I would also like to express my gratitude to Tektronix whose commitment to the continued education of its employees allowed me to complete this thesis.

My thanks also to the early user's of MOSSTAT, Larry Morandi at Tektronix and Ray Sutton at OGC. They both provided constructive criticism that greatly influenced the form and content of MOSSTAT.

Table of Contents

Abstract i

Acknowledgements iv

List Of Figures vi

1. Introduction 1
   1.1 The Design Environment 2
   1.2 Overview of this Thesis 4

2. Error Detection and Isolation 5
   2.1 Typical Errors Found in VLSI Designs 5
   2.2 The Ideal Logic Design Environment 6
   2.3 Other Approaches to Error Detection and Isolation 8
   2.4 Mosstat's Approach to the Problem 12

3. Mosstat — A User's View 14
   3.1 Mosstat Interactive Environment 14
   3.2 Executing Mosstat 16
   3.3 Mosstat Methodology 17

4. Internal Structure of Mosstat 19
   4.1 Initialization Module 19
   4.2 Interactive Module 21

5. Data Structures and Algorithms 24
   5.1 Data Structures 25
   5.2 Memory Allocation 32
   5.3 Classify Gates Command 34

6. Evaluation of Mosstat as a Design Aid 40
   6.1 PRSC 40
   6.2 Objmap Map Chip 41
   6.3 Conclusion 43

7. Conclusions and Future Work 45
   7.1 Additional Work on Mosstat 45
   7.2 Future Static Analysis Tools 47

References 49

Appendix A: Mosstat User's Manual 51

Appendix B: The Grammar for a Boolean Expression 75

Appendix C: Glossary of Terms 79

Appendix D: Mosstat Source Files 83

Biographical Note 87

List of Figures

Figure 1.1: VLSI Design Tools 3
Figure 2.1: Tool Functionality Chart 13
Figure 3.1: Query Commands 15
Figure 3.2: Rule Check Commands 16
Figure 3.3: Miscellaneous Commands 17
Figure 5.1: Node Structures and Hash Table 25
Figure 5.2: Node Structures and Linked Transistor Structures 26
Figure 5.3: Transistor Structure and Node Structures 26
Figure 5.4: Contents of Transistor, Node and Gate Structures 27
Figure 5.5: Structure of WorkSpaces 29
Figure 5.6: Data Structures for Four Distinct Nodes 30
Figure 5.7: Data Structures for Three Aliases 31
Figure 5.8: CMOS Logic Gate 35
Figure 5.9: Logic Gate Data Structures 36
Figure 5.10: Dynamic Logic Gate 38
Figure 6.1: Command Usefulness 41
Figure A.1: Static CMOS Logic Gate 63
Figure A.2: Dynamic CMOS Logic Gate 63

1. Introduction

Error detection and isolation in circuit design is a subject that has generated a great deal of discussion in the last few years. There are an ever-increasing number of approaches to error detection and an even greater number of tools. The approach chosen is strongly is strongly influenced by a number of previous VLSI CAD tools. In addition, a few ideas have been borrowed from some software tools that are not typically used in VLSI CAD.

Static analysis methods have been pursued because this is an approach that needs to be further exploited. I propose that a large class of problems can be found with static analysis and they can be found at a much lower cost than with other types of analysis methods.

The basic problem with existing static analysis tools is that they are primarily batch in nature. This leads to three problems:

1. Single errors have a tendency to 'cascade' where the first error triggers new ones that would not have appeared if the first error had not occurred. Thus the output may contain several hundred or even thousands of error statements when an error is detected. This makes it easy to detect errors but difficult to actually isolate them. Somewhere in the output will be the one or two error statements which actually isolate the error in the circuit description.

2. In an effort to deal with problem number one, past tools have gone out of their way to keep their output cryptic. Most all error statements need to be evaluated by the designer and the brevity of the error statements may make this task difficult.

3. The errors may occur at nodes whose names were automatically generated by an extractor and so they may be hard to identify. The user has no ability to look at the circuit that was actually extracted, so he or she must extrapolate from the source circuit description to the actual circuit transistor list. A ratio error at node 2346 is of little use until the user identifies the location of the node in question.

MOSSTAT, a static analysis tool that has been developed in an effort to overcome some of the shortcomings of the currently available static rule checkers, will be presented. MOSSTAT will support NMOS and CMOS technologies and will be interactive in nature.

The interactive nature of MOSSTAT will allow the user to choose the rules to be checked and the order that they are to be evaluated in. The results of each rule check will not generate a list of error messages but rather will be stored in a 'database'. The user is supplied with a 'query language' that enables him to retrieve the results of the rule checks from the data base. This same 'query language' also allows the user to retrieve other calculated information about the circuit such as connectivity, transistor sizes and node capacitance. The primary goal of all this interactivity is to make it easier for the user to evaluate the results of rule checks in the context of his particular design.

1.1 The Design Environment

MOSSTAT has been designed to be used as a part of a specific design methodology. This strategy only addresses the logical design of a custom VLSI circuit and does not address higher level issues such as the architecture aspects of the design. Although this strategy is driven by a 'top down' approach to design, it acknowledges the need for 'bottom up' interaction as well. Below is a brief description of the steps involved in this design methodology:

1. Define and partition the design with a high level description.
2. Model the design as a network of transistors.
    a) Create the network of transistors.
    b) Add estimated node capacitances.
3. Analyze the design with static methods.
4. Simulate the design to verify functionality and performance.
    a) Create a set of input vectors for this network that adequately tests the functionality and performance of the design.
    b) Evaluate the output vectors from the simulation.
5. Implement the physical layout of the design.
    a) Verify compliance with physical design rules.
    b) Extract a transistor network model from the layout.
6. Analyze the extracted physical design with static methods.
7. Simulate the extracted physical design to verify functionality and performance.
    a) Use the input vectors specified in the simulation of the transistor network.
    b) Evaluate the output vectors from the simulation and compare them with the results from earlier simulations.
8. Verify connectivity with the use of netlist comparators.
9. Fabricate the design.
10. Test the design.
    a) Exercise the circuit with simulation stimuli used in simulation of the design model.
    b) Document yield, and discrepancies in function and performance from simulation of the transistor network model.

As we move from step 1 through step 9 the 'cost' of rectifying design errors increases. Investing the time to carefully debug the design at each step is paid back in fewer errors found at a later stage when they are more 'expensive' to distinct, isolate and correct.

MOSSTAT is used twice in this methodology. Both times are immediately preceding simulation of the design. Static analysis is, in general, less 'expensive' than simulation and therefore all possible errors should be isolated with static analysis before the more expensive simulation step is attempted.

All this is consistent with the 'top down' design methodology. In order to correctly partition the project into modules we must have a pretty fair estimate of the floor plan and the relative sizes of each module. We must also have a good approximation of the floor plan and module size in order to make correct line capacitance estimates for step 2. As we move through the design steps, estimates of the floor plan may change, which may necessitate going back and propagating of these changes down from the top.

There are further, more subtle implications of this design methodology. This methodology developed out of a need to cope with the increasing complexity of integrated circuit design [Mea80]. This methodology imposes constraints on the way circuits are constructed [Noi82]. One can no longer afford to hand-craft each transistor, so 'rules of thumb' are created to aid in the choice of transistor sizes. Clever circuit configurations are avoided in favor of circuits composed under the guidance of composition rules that rule out arbitrary circuits. These new design methodologies have opened up the field of design to a new breed of designer. This new breed of designer may be a sophisticated architect, but need not be a process engineer.

Figure 1.1 is the functional chart for some of the tools and file formats that are supported by the VLSI Design Tools which are distributed by the UW/NW VLSI Consortium. Also included in the chart is MOSSTAT showing how MOSSTAT fits in with the existing tools.

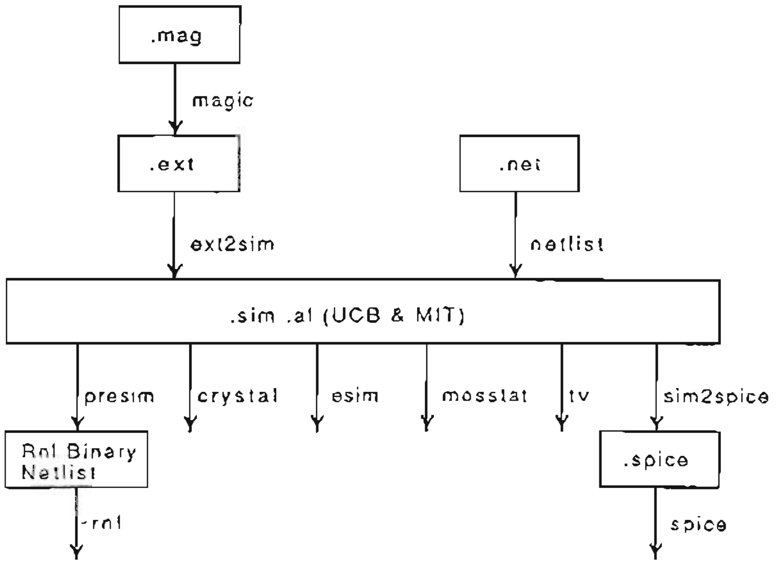


Figure 1.1: VLSI Design Tools

1.2 Overview of this Thesis

MOSSTAT is a static rule checker for custom MOS VLSI circuits. MOSSTAT is interactive in nature which allows the user to specify a number design rule checks to be performed on the design. In addition MOSSTAT contains an expression based query language that allows the user to isolate a variety of design errors. The results of the the rule checks are stored in a database and the query language allows the user to evaluate these results in an orderly fashion.

A number of special terms will be used in this thesis and most of these terms will be defined in the glossary in Appendix C.

This thesis contains seven sections. You are reading Section one, and it contains a introduction to this thesis and some of the motivations behind the design of MOSSTAT. Section two will talk about error detection and isolation and how other tools have been used to detect and isolate errors. Section three will describe MOSSTAT from the user's point of view. This section is not meant to describe the user interface to MOSSTAT in detail and therefore the 'MOSSTAT User's Manual' has been included in Appendix A. Section four will talk about the internal structure and functionality of MOSSTAT. Section five will describe some of the interesting algorithms and data structures used to implement MOSSTAT. Section six will discuss experiences gained from using MOSSTAT and the feed back received from users of MOSSTAT on a simple design. The seventh section will describe the possibilities for future work in the area of static analysis. This work will include features that could be added to MOSSTAT, as well as features that would be a part of a whole new environment.

2. Error Detection and Isolation

This section will cover error detection and isolation. After describing typical errors encountered in VLSI designs, an ideal logic design environment will be proposed. No attempt will be made to address higher levels of design. This ideal design environment is really just a integration of a number of existing tools into one. Each tool has an approach that it takes to detect errors and a particular class of errors that it is designed to detect. The integrated environment allows the user to exploit these different approaches to efficiently detect and isolate problems. Many errors can be detected and isolated by a number of different tools. Due to the differences in the way these tools function, the relative expense of detecting and isolating these errors will be different for each tool. The ideal environment allows the user to isolate the largest number of errors with the least 'expensive' tools. The existing tools will be discussed with special attention paid to how features from of each tool could be propagated into the ideal environment. Next MOSSTAT will be introduced, and the approaches it uses will be discused. Finally, the propagation of these approaches into the ideal environment will be discussed.

Two terms 'error detection' and 'error isolation' will be used that bear some further explanation. Error detection is the ability to detect the presence of an error in a design. An error is detected when a invalid state is detected at a node whose state is being monitored. Error isolation is the ability to pinpoint the source of an error in a design. The source of an error may be a node or transistor that can effect the state of the monitored node. Error isolation is made difficult due to the 'cascading' of errors: a single error may produce a large number of secondary errors. For a tool to be effective at 'error isolation' it must be effective at detecting the error directly. If that is not possible the tool should assist the user in dealing with the secondary errors.

2.1 Typical Errors Found in VLSI Designs

This thesis classifies typical VLSI design errors into three groups:

Logic Errors. A logic error is any error that effects the functional behavior of the design. As designs get larger, errors can become very difficult to detect and even more difficult to isolate. In order to detect an error three things must occur:

1) The input stream must exercise the error fault.

2) The error must be propagated to a node that is being monitored.

3) The error cannot be masked by other errors.

In order to isolate errors the user must start at a point where the error is observed and backtrack until the error is discovered.

Timing Errors. A design may functionally perform according to specification but not perform at the specified speed. These errors are due to either overlooking a possible 'critical path' or getting inaccurate timing information about a known critical path. Due to the expense of accurate timing analysis it can only be performed on a small section of a large circuit. The designer must speculate or use special tools to know on which paths to perform timing analysis. The designer must correctly identify all of the critical paths in order to get correct behavior from the fabricated chips. A timing error is often detected as a logic error, but when the source of the error is isolated it is really a timing error. These errors are typically difficult to detect and and very difficult to isolate.

Implementation Errors. A case could be presented that would support the claim that most errors are really due to implementation errors. We often call them 'dumb' errors, but they are really just due to human nature and a design environment that readily allows the introduction of these errors. Implementation errors can always be detected as either timing or logic errors but what they really are is inconsistencies between the design specification and the design implementation. An omitted connection is the first implementation error that comes to mind. Another example would be an improper connection such as connecting two signal lines out of order. In NMOS ratio errors are common and although CMOS is typically ratio-less it can be contain ratioed logic that has incorrect transistor sizes. By misplacing a well or typing a 'p' instead of an 'e' a designer could end up with a p-type transistor in a path to GND which would mean that a node cannot be set below one threshold drop.

There are several ways that improper use of pass transistors can cause implementation errors. In NMOS if a signal is passed through a pass transistor, it is an error to use this signal to gate another pass transistor. In CMOS a transmission gate must consist of two transistors, one p-type and one e-type, and to omit either one would be an error.

Often a designer may be using design 'rules of thumb' as part of his design methodology. Although enforcement of these rules is not mandatory, violation of these rules should be cause for further investigation. An example of such rules of thumb are the Clocking Rules for Dynamic MOS Systems [Bra86].

2.2 The Ideal Logic Design Environment

When we think about the ease with which we can introduce errors into a design and the difficulties that we have in finding and removing these errors, we begin to appreciate the difficulties of fabricating a correct VLSI design. My primary focus in this discussion will deal with the verification of performance and functionality. As this 'ideal' design environment is proposed no concessions are made to the realities of actually implementing such an environment.

Schematic capture, physical design, simulation, electrical rule checking and design rule checking must all be a part of one integrated environment. The designer must be able to view and edit at multiple levels of abstraction. At a minimum the environment will have to support a schematic, physical and a extracted netlist description of a design simultaneously. The data base to support this environment will have to contain all necessary information to describe the design at multiple levels of abstraction. The consistency of this 'vertical hierarchy' must be automatically enforced by the environment to minimize chance for error.

The design environment needs to contain the following support:

Incremental design rule checking. We already see this feature in many physical layout systems.

Incremental static analysis. Many of the implementation errors mentioned in Section 2.1 could be totally eliminated with this type of support.

Incremental extraction. As a design is entered either physically or schematically the netlist description should also be a part of the data base. This netlist description will have to be hierarchical in nature.

Static Analysis. Some static analysis can only be done in the later stages of the design cycle. An example would be the need to know the nodes that are to be declared as inputs or outputs. Some of this analysis can be an aid in 'critiquing' digital designs, encompassing issues such as functional correctness, design style, operating speed, critical path identification.

Static Timing Analysis. An accurate timing analysis tool will have to be available. It will be easier to use because it is built into the environment. The other analysis tools in the environment will be used to isolate the portions of the circuit that may need this level of analysis.

Functional Simulation. A mixed-mode simulator will almost certainly remain as an important part of the design environment.

The specifications of this 'ideal' design environment have been largely influenced by the evolution of design rule checkers (DRC's) that has occurred over the past several years. At first the physical editors were really just graphical 'box' generators that knew little or nothing of how these boxes interact with one another. The DRC's were batch oriented and we discovered that the cost of fixing physical design violations was extremely high. By making the DRC's part of the physical layout system and by making them incremental and interactive, it became easy to detect design rule violations. Isolation and repair of design rule errors at this early stage of the design cycle is extremely painless.

We know that as the design cycle proceeds the cost of detecting, isolating and correcting errors increases. It is for that reason that we must also make every effort to isolate these errors at the earliest possible step in the design cycle. Like the interactive design rule checker we must make an effort to prevent these errors from ever getting into the design. When errors do get into the design we must detect them as early as possible. Unfortunately not all implementation errors can be uncovered incrementally. Static analysis can then be used to uncover many remaining implementation errors.

2.3 Other Approaches to Error Detection and Isolation

In the past there have been primarily two approaches to identifying and isolating errors in the specification of a VLSI design. The most first and most common method is through simulation. Simulation is used extensively because it theoretically detects a very large class of errors. If the functionality of a circuit is fully exercised with a gate level simulator and you have achieved reasonably good fault coverage, you can feel quite confident that all implementation errors have been uncovered. Furthermore if you have speed constraints on your design, with the right tools you can theoretically get accurate timing analysis.

The second method of error detection and isolation is through static analysis. Usually static analysis tools are written to isolate a particular class of errors. The very nature of static analysis is much more simplistic than dynamic analysis (simulation) and is therefore less costly to perform. Due to the fact that dynamic analysis deals with the functional and timing behavior of the circuit, it is effective at error detection but it has a tendency to be ineffective at isolating errors. It is for these reasons that errors that can be found with a reasonable amount of static analysis can be detected and isolated at a lower cost.

2.3.1 Simulation Techniques

SPICE is a general-purpose circuit-simulation program for nonlinear dc, nonlinear transient, and linear ac analysis[Spi85]. There are also a number of SPICE 'clones' in existence, that seek to build upon the well proven models developed in SPICE. SPICE has proven to give extremely accurate timing analysis when the proper models are used.

The disadvantage of SPICE is that it can not simulate a large digital circuit. This is due to the fact that as the number of gates and connections increase, the computational requirements increase exponentially. Designers will always have to resort to simulating smaller functional units, and making assumptions about the interaction between these units. Due to the expense of SPICE simulations it is not an effective tool for the detection and isolation of implementation errors. Because of the expense of SPICE simulations it is also not a effective tool for detecting or isolating timing errors unless the critical paths are first identified with some other tool. SPICE is not a logic simulator and therefore is not appropriate for uncovering logic errors. The batch nature or SPICE make it impossible to give any real support to error isolation. SPICE type simulators will always have a place in a design environment, they will remain the primary method of simulation when accurate timing analysis is needed.

Three commonly used logic simulators are MOSSIM [Bry83], RNL and ESIM [Ter83]. MOSSIM is a switch-level simulator for MOS technology. MOSSIM uses a switch-level model that consists of a set of nodes connected by transistor switches. ESIM is an event-driven switch-level simulator for NMOS and CMOS transistor circuits. A switch-level simulator bases its predictions on an abstraction of the actual circuit. The switch-level model proposed in ESIM is able to handle the bidirectional nature of MOS transistors much more successfully than a gate-level model. RNL attempts to give timing information for digital MOS circuits. It is an event-driven switch-level simulator that uses a simple RC (resistance capacitance) model of the circuit to estimate node transition times and to estimate the effects of charge sharing. This simple model is quite efficient however, in certain circumstances it can give inaccurate timing information particularly in CMOS designs. RNL suplies the user with a simple LISP interpreter as a user interface. This allows both interactive simulation and the programming of complex functional simulations.

Some simulators can give timing information but only if the modules are first 'parameterized' using SPICE. ESIM is effective at detecting logic errors but does no timing analysis. RNL, can give some timing information but it uses a crude transistor model that can in certain situations be inaccurate. Logic simulation is an important portion of the design process today and it will continue to be in the future.

The first problem with depending on logic simulation to detect and isolate errors is that getting 100% error coverage can be difficult or impossible. The second problem with using a logic simulator for error detection and isolation is that error isolation is given little or no support. We can detect an error at a monitored node but the error could have occurred at any number of nodes 'upstream' from the monitored node. The exception is RNL which gives the user the ability to stop the simulation at any point. The user can then interactively query for information about connectivity and present state information for any node in the netlist. The third problem is that logic simulation is expensive in both computer and user time. The user must generate a set of input vectors and then must analyze the output vectors and both of these chores are error prone. Therefore we need to investigate ways to reduce the dependencies upon simulation to detect and isolate error in circuit design.

2.3.2 Static Analysis Tools

The appearance of static analysis tools was due in the most part to the shortcomings of the existing method of using logic simulation to detect all errors in a VLSI circuit. The problem with any tool that uses static analysis is that it cannot detect logic errors. It can, however, detect timing errors and many implementation errors. Each of these analysis tools were designed to detect a particular class of errors. Some look for timing errors others look for implementation errors. These are all errors that can be detected with SPICE or a logic simulator. These tools strive to detect and isolate these errors at a lower cost. They also typically are good at isolating errors. Some of these tools are designed to be used on a particular class of designs.

2.3.2.1 Timing Analyzers

TV is a timing analyzer for NMOS designs [TV83]. Based on the circuit description obtained from existing circuit extractors, TV determines the minimum clock duty and cycle times, and verifies that the circuit obeys the MIPS clocking methodology. TV stresses fast running time, small input requirements, and the ability to offer a user valuable advise. TV is can be effective at detecting and isolating timing errors.

CRYSTAL is a program that analyzes the performance of VLSI circuits [Cry83]. In addition to the transistor netlist the user supplies a few lines of text to aid CRYSTAL in determining current direction. CRYSTAL then determines how long each clock phase must be and prints out information about the portions of the circuit that generate the worst delays.

Both these tools aid in performance tuning by pointing out paths that limit clock speed. Critical path identification is becoming both increasingly difficult and extremely important. Circuits become so large that the designer rarely knows all the paths that could become critical. Both TV and CRYSTAL must determine the direction of current flow through the transistors in the circuit. Although CRYSTAL does allow some analysis of input vectors, they both do the majority of their calculations independent of input data. CRYSTAL attempts to determine the current direction in critical pass transistors and it allows the user to specify the direction when it is unable to. TV has a more sophisticated method for determining current direction, but it attempts to do so without input from the user.

The detraction of these type of tools is that they require the user to learn a whole new tool. CRYSTAL can be particularly difficult to supply the needed current direction information. A critical path tool such TV or CRYSTAL could play an important role in the ideal design environment. The need to interactively specify current direction would be much less of a stumbling block if it were a part of a consistent user interface and if the input could be input interactively rather then including it in the physical description.

2.3.2.2 Electrical Rule Analysis

Static Analysis Program (STAT) is an analysis tool for NMOS designs [Bak80]. It does a batch analysis of a circuit with no input from the user. The circuit description STAT reads from is a transistor list containing information about transistors sizes (resistance) , transistor connectivity and line capacitance. STAT will infer the existence of inputs by looking for lightning arrestors that are normally found on input pads. STAT will classify the transistors, calculate threshold drops and do ratio checks on 'nand', 'nor' and 'inverter' gates. It also checks for nodes which cannot be set high or cannot be set low and finds all nodes which cannot be effected by the inputs or cannot effect any outputs. For each analysis it does STAT has a set of boundary conditions, and if any analysis yields results that are not within these boundaries, STAT will issue a error message.

Electrical Rule Checker (ERC) is another example of an analysis tool for NMOS design[Pen82]. ERC reads a transistor-level circuit description of an NMOS chip produced by MEXTRA and a designer-supplied list of inputs and outputs. It produces a listing of possible electrical rule violations similar to those detected by STAT, a cross-reference of nodes and transistors, and CIF code to physically display the locations of the electrical rule violations.

A big shortcoming of ERC is that it cannot be used to analyze a circuit until the circuit is physically laid out. To make corrections at this stage of the design is simply too expensive. Actually the concept of being able to graphically review the errors generated by a rule checker is very sound but the designer needs to be able to review the errors at the schematic or netlist level first, before the physical design is begun.

ERC does a number of analyses on the circuit design. The user can control what types of analysis are to be carried out by specifying at run time a list of commands. This is done with command line options that allow the user to enable or disable certain rule checks.

ERC's ability to limit the number of rule checks done in a given execution results from a need to deal with the potentially overwhelming number of error messages that can come from a circuit in its early stages of development. It also increases the speed at which a circuit can be evaluated. ERC also sought to make it easier to isolate errors by graphically displaying them.

Both STAT and ERC were designed to detect implementation errors rather that timing or logic errors. Although these errors could be isolated with logic simulation techniques, STAT and ERC strive to detect and isolate these errors at a lower cost.

2.3.2.3 Rule- Based Tools

Several new tools have emerged using 'expert system technology' to provide meaningful feedback to circuit designers on the quality of their design in a manner similar to the critique provided by an experienced designer. Two examples of these tools are CRITTER[Kel84] and RUBICC[Lob84]. CRITTER is an exploratory prototype design aid, built using artificial intelligence technology, to aid in 'critiquing' digital circuit designs. The issues addressed are functional correctness, operating speed, timing robustness, and circuit sensitivity to changes in device parameters. Using artificial intelligence techniques CRITTER attempts to take on some traditionally human judgement tasks. RUBICC is an expert system written to critique the design of VLSI circuits at the transistor level. RUBICC performs its critique by 'reasoning' about the design using rules contained in its knowledge base. Both of these systems perform their critique without the use of simulation. An important part of these systems is their ability to isolate errors as they are detected. This approach can give static analysis the ability to detect and isolate timing and logic errors without the use of simulation. This approach to analysis of designs has a great deal of potential for future design environments.

2.4 MOSSTAT's Approach to the Problem

MOS Static Analysis Program (MOSSTAT) is an analysis tool for NMOS and CMOS designs. MOSSTAT does many of the same types of analyses on a design as ERC or STAT. The primary difference is the interactive nature of MOSSTAT. MOSSTAT creates an internal database that contains all the information retrieved from a netlist description of the circuit. MOSSTAT allows the user to interactively select from a number of different static rule checks to verify a design. The results of the rule checks are stored in the database. MOSSTAT provides the user with a rich set of commands for interactively retrieving information from the 'database' such as connectivity, transistor sizes or the results of a rule check. This query language can be a powerful tool for the isolation of errors in a design.

Like ERC and STAT, MOSSTAT is effective at the detection of implementation errors. MOSSTAT will not require extra circuitry to be added to imply the existence of inputs or outputs. Unlike ERC and STAT, MOSSTAT can be used on CMOS as well as NMOS designs. Due to MOSSTAT's interactivity it is much more effective at isolating errors.

Like TV and CRYSTAL, MOSSTAT can be used to isolate critical paths although it not as effective due to the fact that MOSSTAT assumes all pass transistors (transmission gates) are turned off. This is a severe shortcoming, but it would be easy to fix in future versions.

MOSSTAT includes a number of unique features. It can identify the nodes that function as logic gates. Once the logic gates have been identified the the user can easily analyze the timing, ratios or functions of the logic gates. These features allow the user to enforce some design constraints on the logic gates.

Unlike CRITTER and RUBICC MOSSTAT has not used a expert system approach. Such an approach would make it into the ideal static analysis tool. At this time it was considered to be outside of the scope of this project.

MOSSTAT will detect and isolate many implementation errors, and because it does it without simulations it can do it at a lower cost than either RNL or SPICE. MOSSTAT will detect and isolate many implementation errors, and because the analysis is static in nature it can do its analysis at a lower cost than either RNL or SPICE. Because MOSSAT does its analysis statically the user does not need to specify the input vectors nor does the user have to generate expected output vectors. In addition static analysis is less computational than dynamic analysis and therefore runs much faster. MOSSTAT can provide some assistance at finding possible critical paths, but ultimately these paths will have to be verified with a timing analysis tool such as SPICE or RNL.

Performance is a key priority for MOSSTAT. It is paramount for the interactive commands in MOSSTAT to run quickly. Any advantages of an interactive environment are quickly lost if some of commands take longer that a few seconds to execute.

In summary, the table in Figure 2.1 gives a brief comparison of the verification capabilities of each of the tools described. No star means that the tool gives little or no support for that particular function. One star means that the tool gives some support for that particular function. Two stars means that the tool gives excellent support for that particular function.

Functionality Chart						
	Functional Errors		Timing Errors		Implementation Errors	
Tool	Detection	Isolation	Detection	Isolation	Detection	Isolation
SPICE			**	*		
RNL	**	*	*	*	**	*
ESIM	**			**		
MOSSIM	**	*				
CRYSTAL			**	**		
TV			**	**		
ERC					**	*
STAT					**	
MOSSTAT			*	*	**	**

Figure 2.1: Tool Functionality Chart

3. Mosstat — A User's View

This section briefly describes the functional goals of MOSSTAT, and it avoids any description of syntax. If more information is desired about the syntax of MOSSTAT, Appendix A contains a user's manual that describes the user interface in detail.

MOSSTAT is an interactive static rule checker for NMOS and CMOS VLSI circuit designs. MOSSTAT allows the user to interactively select from a number of different static rule checks to verify a design. MOSSTAT initially creates a database that will contain all the information retrieved from a netlist description of the circuit. MOSSTAT provides the user with a rich set of commands for retrieving information contained in the database such as connectivity, transistor sizes or node capacitance.

3.1 Mosstat Interactive Environment

The commands in the interactive environment fall into three categories. The first containing 'query' commands that allow the user to create workspaces for user selected subsets of the database. The second contains 'rule check' commands that allow the user to retrieve or manipulate information from the database. The third category contains 'miscellaneous' commands that don't fit into one of the other categories.

3.1.1 Query Commands

All query commands access the database through 'workspaces'. A workspace is just a list of elements extracted from the database. There are three types of elements associated with each workspace: TRANS, NODE or GATE. The workspace is of type TRANS if it contains a list of transistor elements. The workspace is of type NODE if it contains a list of node elements. The workspace is of type GATE if it contains a list of nodes that have been classified as logic gates.

There are two classes of workspaces 'system' and 'user'. There are initially two 'system' workspaces: one that is a list of all the transistors in the netlist, and another that is a list of all the nodes in the workspace. A 'system' workspace is created by MOSSTAT without intervention by the user. A 'user' workspace is created with the 'for' command and is a subset of either a 'system' workspace or another 'user' workspace.

Each element in a workspace contains a number of parameters. Transistor length and transistor type are examples of transistor parameters. Node name and node capacitance are examples of node parameters.

The boolean expression in a 'for' command may contain constants or parameters. This expression is evaluated on each element in the source workspace. If the expression evaluates to TRUE then the element is added to the destination workspace. The source workspace may be a user or system workspace, but the destination workspace can only be a user workspace.

The 'count' command will count and categorize the elements in a workspace. The 'print' command will print out the contents of the workspace to either the terminal or to an external file.

Figure 3.1 lists all the query commands and gives a brief description of each. For a more detailed description see the user's manual in Appendix A.

Query Commands	
count	Count and classify a workspace.
for	Create a workspace from another.
print	Print the contents of a workspace.

Figure 3.1

3.1.2 Rule Check Commands

The rule check commands are a set of rule checks that can be performed on the circuit. The results of these rule checks are added to the database. The information can then be retrieved from the database with the query commands.

The 'classifygates' command will identify nodes that can be classified logic gates. It will then classify these logic gates. A logic gate is a gate in the boolean sense, it may be an inverter or a complex nand-nor function. A system workspace is created that contains a list of all the nodes that are classified as logic gates. A workspace can then be created using a query command that contains the logic gates with, for example, the rise and fall times in any given range.

The 'effects' command will mark all nodes that can be effected by the inputs and all nodes that can effect the outputs.

The 'propagate' command marks all the nodes that can be set high or low.

The 'threshold' command calculates all the threshold drops and threshold rises at each node. The concept of a threshold drop is undoubtably familiar, but the concept of a threshold rise may need further explanation.

A threshold rise is to a p-transistor what a threshold drop is to an e-transistor. When a p-transistor is turned on (its gate is low) it can only pull its drain down to within one threshold rise of it gate voltage. Thus the voltage range for a node driven solely by a p-transistor is 1-5 volts.

Figure 3.2 list all the rule check commands and gives a brief description of the each. For a more detailed description see the user's manual in Appendix A.

Rule Check Commands	
classifygates	Classify the logic gates.
effects	Find nodes effected by an input and that effect an output.
propagate	Mark nodes that can be set high or low.
thresholds	Calculate threshold drops and rises

Figure 3.2

3.1.3 Miscellaneous Commands

Miscellaneous commands can be used to declare inputs, outputs or clocks. They also provide access to a help facility and a few commands to aid in debugging. Interactive commands in an external file can be executed with the 'source' command.

Figure 3.3 lists all the miscellaneous commands and gives a brief description of each. For a more detailed description see the user's manual in Appendix A.

3.2 Executing Mosstat

When MOSSTAT is executed it reads in a netlist description which may optionally contain a list of node aliases. The information from the netlist description is installed in the internal database. All the distinct nodes that are aliased together are collapsed into one element in the database. As this database is created two system workspaces are created: the 'nodews' which is a list of node elements in the database, and the 'transws' which is a list of transistor elements in the database. There is also a third system workspace which will be discussed in section 5. MOSSTAT then checks for the existence of VDD and GND. MOSSTAT is now ready to receive interactive commands from the user.

Miscellaneous Commands	
alias	List all aliases for a node.
clocks	Declare nodes to be clocks.
help	Help.
inputs	Declare nodes to be inputs.
outputs	Declare node to be outputs.
paths	Display paths to VDD and GND.
quit	Normal exit from MOSSTAT.
setalias	Create aliases.
setpf	Set print flags.
unsetpf	Unset print flags.
source	Execute command in file name.
system	Execute shell command.

Figure 3.3

Next the user runs commands to declare all the clocks, inputs and outputs. The user is now allowed to run rule check commands or query commands in any order that is convenient for the user.

3.3 Mosstat Methodology

The goal of a static analysis tool such as MOSSTAT is to detect and isolate a specific class of errors and do it at the lowest possible cost. MOSSTAT provides the user with a number of built-in rule checks which can aid in the detection and isolation of errors. The ability to selectively execute these rule checks allows the user to control the order in which rule checks are executed. The real heart of the productivity of MOSSTAT is the 'for' command. The 'for' command implements a powerful expression based interactive query language. This expression based language enables the user to execute a simple rule check, then use the query commands to detect and isolate errors found by the first rule check. These errors are found before the next rule check has to be executed. In addition the 'for' command also allows the user to enforce many rules that are not specifically checked for the 'rule check' commands.

To start, the user uses the 'count' command to isolate certain categories of nodes or transistors that need to be investigated for possible errors. After the inputs, outputs and clocks have been declared the user can begin by performing some simple rule checks. The best rule checks to start with are the 'propagate' and 'effects' commands. These rule checks isolate a large number of errors due to connections not made or to improper connections. The next rule check should be 'thresholds'. An invalid connection error will often be detected as an invalid threshold rise or threshold drop. An invalid threshold rise or drop is an example of an error that could be detected by a logic simulator such as RNL. Due to the simplicity of the static analysis needed to detect these errors they should be eliminated before simulation begins. The last rule checks and the most sophisticated is 'classifygates'.

The reason this particular order was chosen is that these rule checks are executed in the same order as their relative complexity or cost. The user should attempt to detect and isolate as many errors as possible with the more simplistic rule checks before he attempts the more expensive rule checks. In addition it makes no sense to execute complicated rule checks such as 'classifygates' until we get most of the simpler bugs out of the system. In fact it is doubtful that classifygates would be able get useful results from a circuit that contains invalid connections.

When a node has been classified as a logic gate there are number of further checks that can be performed with the 'for' command. Critical path analysis or resistance ratio calculations can be performed on any logic gate.

For a more thorough discussion the methods that could be used with MOSSTAT see section four of the user's manual in Appendix A.

4. Internal Structure of Mosstat

This section contains a description of the internal structure of MOSSTAT. The structural aspects of the database will not be described, as that will be covered in detail in section five. The goal of this section is to familiarize the reader with the structure of the code that implements MOSSTAT.

The program is divided into two modules according to function. The first module is the initialization module. It parses the 'simfile', and the 'aliasfile', and creates the internal database representation of the circuit. These actions are performed without any interaction with the user. The second module, the interactive module, implements the MOSSTAT Interactive Environment. This module is highly interactive and allows the user to retrieve information from the database, create user workspaces and execute rule check commands.

Appendix D contains a brief description of MOSSTAT's source files. Detailed documentation has been included in the source, which is available in 'Mosstat: An Interactive Static Rule Checker for MOS VLSI Designs, Source Code'[Joh86].

4.1 Initialization Module

This module performs three functions:

1) Parse the 'simfile' and the 'aliasfile'.

2) Create the database.

3) Add aliases to the database.

4.1.1 Parsing the Simfile and the Aliasfile

MOSSTAT will accept simfiles that are in either of two formats. The name 'simfile' is derived from the mandatory name extension '.sim'. The format of the file will depend on the tool that was used to create the 'simfile'. The two formats will hereafter be referred to as the 'UCB' format and the 'MIT' format. They are described more completely in the simfile man page [UW85]. Each line of the simfile (and the aliasfile described below) contains a record whose type is determined by the first character of the line.

The only valid records in the simfile are the information, transistor, input redirection and capacitance records. The only valid record in the aliasfile is an equivalence record. All other records will be discarded after issuing a a warning.

There are several undocumented conventions that are observed by MOSSTAT and all other tools that read and generate simfiles. First, the equivalence records are not contained in the simfile but are contained in a second file, the aliasfile. The equivalence record allows the extractor to connect distinct nodes or create name aliases to nodes. The aliasfile will have the same root name as the simfile but will have the name extension '.al'. The second undocumented convention is that the information record is assumed to be the first record of the simfile and all other information records will be ignored. The information record specifies the format, scale and technology of the simfile.

For parsing the simfile, a recursive-descent parser was used rather than using a parser generator. There are several reasons for this decision. First, the format of the simfile is quite simple and its definition is very stable. Second, the simfile can be very large and performance is a prominent issue. Third, the scanner could take advantage of the 'record oriented' syntax of the simfile. And lastly, since the simfiles and aliasfiles are generated by other programs, they can be assumed to be syntacticly error free; invalid records are just discarded.

The same scanner is used for both the simfile and the aliasfile. The scanner reads the record and stores each token in the record into an array. Once the information record has been parsed, all records are assumed to be in the format specified by the information record. After the simfile has been parsed, MOSSTAT will scan and parse the aliasfile.

4.1.2 Create the Data Base

The simfile is just a list of transistors and the nodes to which they are connected. As the simfile is being parsed, MOSSTAT makes a unique entry in the database for each transistor. In addition, an unique entry is made for each node in the simfile. Each transistor record represents a unique transistor that is connected to three nodes. A node may be connected to an arbitrary number of transistors.

When the entire simfile has been parsed the database will consist of two system workspaces: *nodews* and *transws*. The *nodews* contains a list of node elements that contain the capacitance due to area, the capacitance due to transistor gates and a list of all the transistors that are connected to the node. The *transws* contains a list of transistor elements that contain the length, width and area of the transistor and a list of the nodes that are connected to the transistor.

4.1.3 Add Aliases to Data Base

The equivalence record in an aliasfile is a list of N nodes, for which nodes 2 through N are to be connected to node 1. Since there can be only one entry in the data base for each node, node entries 2 through N in the list must be 'collapsed' into the first entry in the list. The first entry in the list then becomes the primary node and all other entries in the list are aliases that refer to the primary node. The primary node is now the 'sum' of all the nodes that are aliased to it. A thorough discussion of how aliases are implemented is contained in section five.

When the aliasfile has been parsed, the nodews must be compressed to fill in vacancies due to nodes being connected. The primary node is now the 'sum' of all the nodes that are aliased to it. There can be any number of aliases for each node entry in the database but each primary node entry must be unique.

4.2 Interactive Module

The interactive module performs three functions:

1) Parse the interactive commands.

2) Execute rule check commands and query commands other than the 'for' query command.

3) Evaluate the boolean expression in the 'for' command.

4.2.1 Scanning and Parsing the Interactive Commands

The approach of treating the user input to MOSSTAT as a language and applying compiler technology to construct an interpreter has several benefits:

1) *Ease of implementation*. The language is of sufficient complexity to merit the use of compiler technology.

2) *Language evolution*. A syntax-directed definition facilitates changes in the input language. It seemed that defining a language and building a compiler for it using a compiler generator seemed the only sensible approach.

The scanner and parser for the interactive commands have been generated by LEX and YACC respectively. Both LEX and YACC are distributed with UNIX 4.2bsd†.

A complete definition of the interactive command language grammar is contained in the YACC description in the sources for MOSSTAT described in Appendix D. The grammar for a 'BooleanExpression' is described in Appendix B.

The scanner generates a token stream and had to have the ability to backtrack. This backtracking was needed due to the fact that the language must allow a node name to contain any printable character. This is done by only allowing the scanner to return a

† UNIX is a trademark of Bell Laboratories.

node name as a token when it is scanning a StringExpression. For a number of reasons the scanner generated by LEX contains a large number of transitions and states. These reasons are outlined in Section 7.2.2. Consequently LEX had to be recompiled with a number of changes made to accommodate the large number of states and transitions needed.

The parser parses the token stream from the scanner. As this is done the type rules are enforced as specified for the BooleanExpression that is described in Appendix B. If any errors or the interrupt character (usually ^C) are encountered, the input stream is flushed and MOSSTAT returns to the prompt.

4.2.2 Execute the Rule Check and Query Commands

All of the rule check commands are easy to parse and are executed by a procedure called from within the parser as the command is parsed. All the query commands with the exception of the 'for' command are executed in the same fashion. The data structure allows random access of nodes by hashing of the node names. If a miscellaneous command contains a list of node names then the node elements in the data are accessed directly through the hash table.

4.2.3 Execute the 'for' Command

The 'for' command contains a BooleanExpression that must be evaluated on each element in the specified workspace. The grammar for a BooleanExpression is described in Appendix B. A 'for' command also specifies a source and a destination workspace. Each expression is typed. Expression type must be consistent with the type of the source workspace. Each expression is made up of two or more operands. These operands must be of a consistent type. These operands may be numeric constants (float or integer), string constants, or parameters. Parameters are the variables that are contained in each node, transistor or gate element. Each parameter is typed. A parameter in a transistor element is of type TRANS. A parameter in a node element is of type NODE. A parameter in a gate element is of type GATE.. The type of an expression is determined by passing the type of all operands up through the parse tree.

An expression is of type TRANS if every parameter in it is of type TRANS. An expression is of type NODE if every parameter in it is of type NODE. An expression is of type GATE if at least one parameter is of type GATE and the rest are of type GATE or NODE. An expression of a given type can only be evaluated on a workspace of the same type. The only exception to this rule is that an expression of type NODE may be evaluated on a workspace of type GATE. A workspace that contains transistor elements is of type TRANS. A workspace that contains node elements is of type NODE. A workspace that contains gate elements is of type GATE.

As the boolean expression is parsed, a RPN (reverse polish notation) representation of the expression is pushed onto a stack. If no type violations occur then this ‘expression stack’ is then ‘evaluated’ on each element in the source workspace. If the expression evaluates to true then the element is added to the destination workspace. The destination workspace will inherit the type of the source workspace.

5. Data Structures and Algorithms

The focal point of the design of MOSSTAT was its interactive nature. In order for this interactive environment to be productive, no command can take more than two seconds to execute. This had to hold true for designs up to 10,000 transistors, and for designs of greater than 10,000 transistors the response time had to be linear with respect to the number of transistors. In view of the performance of past static analysis tools it was obvious that this was not going to be an easy task.

The first step in the design was to specify the user interface. The next step was the specification of the data structures. Performance and clarity were the key issues when specifying the data structures. Much of this performance came at the cost of increased size and complexity of the data structures. In view of the current tendencies toward larger systems memories, this seemed an appropriate tradeoff.

Locality of reference is another important issue that needs to be addressed. MOSSTAT will tend to have a very large data set and although ideally it will only be used on a system that has enough real memory to allow the entire data set to be resident in memory this may not always be the case. It may used on a workstation that doesn't have enough real memory or it may be on a multiuser system that continually 'swaps out' pages every time the user pauses for even a few seconds. In either case increased locality of reference will minimize the effects of paging.

MOSSTAT is written in 'C' and is currently running on DEC VAX 11/780 under UNIX 4.2bsd. Performance figures will be quoted for that machine†.

Section five covers three topics. The first describes the data structures that implement the workspaces and aliases in MOSSTAT and why these particular data structures were used. The second topic is memory allocation in MOSSTAT and why MOSSTAT handles its own memory allocation. The third topic is the implementation of the 'classifygates' command. The 'classifygates' command is divided into two sections: the first presents the methodology that was used to identify the 'logicgates' and the second section presents the methodology that was to classify the 'logicgates'.

† The scanner generated by LEX is large and although it is valid 'C', the compiler will abort when it attempts to compile the scanner. This is due to the limitation that UNIX shells put on the stack size for a process (typically .5 MByte).

5.1 Data Structures

Workspaces and aliases are two features of MOSSTAT whose implementation is of particular interest. Before these implementations can be described in detail the basic data elements that make up the database must be presented. These data elements use standard methods that are identical to other, similar tools†.

The basic data elements are structures in 'C'. There is a structure (NODESTRUCT) that describes each node and alias in the netlist description. There is a structure (TRANSSTRUCT) that describes each transistor in the netlist description.

The NODESTRUCTs are accessible through a hash table that is hashed on the node name (Figure 5.1). A linked list is created to handle NODESTRUCTs with duplicate hash entries.

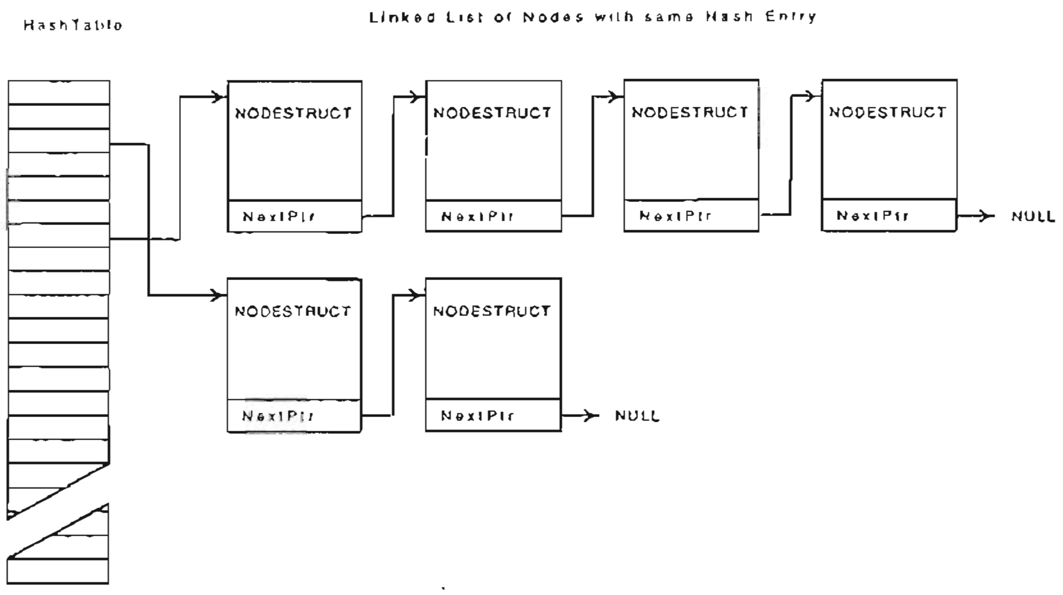


Figure 5.1: Node Structures and Hash Table

The NODESTRUCT contains pointers to three linked lists (Figure 5.2). The first is a list of TRANSSTRUCTs that describe the transistors that have their gates connected to the node described by the NODESTRUCT. The second and third lists are for the TRANSSTRUCTs that have their sources and drains connected to the node described by the NODESTRUCT.

† RNL [UW85], STAT [Bak80], ERC [Pen82]

The TRANSSTRUCT structure contains three pointers (Figure 5.3) one for each of the NODESTRUCTs that describe the nodes connected to the gate, source and drain of the transistor described by the TRANSSTRUCT.

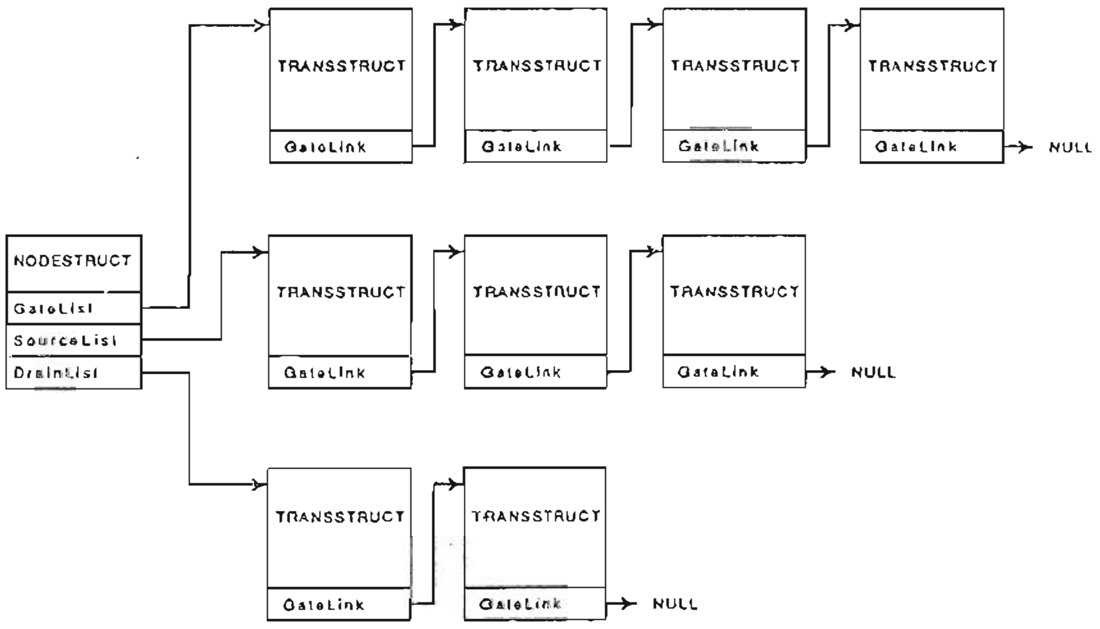


Figure 5.2: Node Structures and Linked Transistor Structures

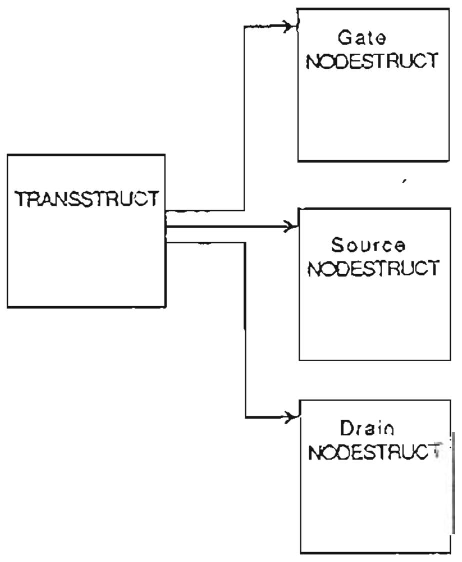


Figure 5.3: Transistor Structure and Node Structures

The contents of each structure is displayed in Figure 5.4. Each entry in the structure is either a pointer or a parameter. All pointer entries are marked with a '*'. The GATES-TRUCT is a structure associated with nodes that have been classified as logicgates and will be introduced in Section 5.3.4.

NODESTRUCT	TRANSSTRUCT	GATESTRUCT
Flags	Type	Flags
ThreshDrop	Direction	MaxRtoGnd
ThreshRise	Length	MinRtoGnd
GateCap	Width	MaxRtoVdd
AreaCap	Area	MinRtoVdd
NodeName*	GateLink*	NetCap
AliasList*	SourceLink*	PathToGnd*
AliasLink*	DrainLink*	PathToVdd*
GateList*	SourcePtr*	
DrainList*	DrainPtr*	
NextPtr*	PathToGnd*	
LogicGatePtr*	PathToVdd*	

Figure 5.4: Contents of Transistor, Node and Gate Structures

These are very effective data structures for three reasons. First, the connectivity of the netlist has been captured. Second the hash table allows quick random access to the NODESTRUCTs. And third the availability of the pointer to connected nodes and transistors allows MOSSTAT to quickly 'traverse' the network.

These data structures however are not efficient for other types of access. As an example, it is not possible to directly access the TRANSSTRUCTs except by way of the NODES-TRUCTs. Previous tools chose to live with the performance penality of these data structures rather then create more complicated data structures. MOSSTAT however, needed more sophisticated data structures in order to meet the initial performance specifications of the interactive environment.

5.1.1 Workspace Data Structures

The concept of using expressions to create workspaces is not new. INGRES [Eps77], a database management system is just one example of another system that uses expressions in this manner. The heart of the INGRES query language was the ability to create workspaces that only contained the information specified by the expression. These workspaces made it easy to evaluate information in the database, because the user had the ability to filter out the data that was not relevant to the query.

When an analysis needs to be performed on all the nodes in the design, the access could be done by a sequential access of all the entries in the hash table. The problem is that a sequential progression through the hash table approximates a random access of the data space. Even worse, when an analysis is to be performed on transistors, the only access to the TRANSSTRUCTs is by indirection through the NODESTRUCTs. In both these cases the access method is easy to specify but is inefficient. What is needed is a method to access the database that takes full advantage of any 'locality of reference' in the data space. In addition the method of access should be simple to implement.

To learn about exploiting locality of reference we must look at the way the netlist description is created. There are two commonly used extractors: MEXTRA and the one built into MAGIC [UCB85]. MEXTRA starts by flattening the physical design and then generating a transistor list that starts from one end of the design and proceeds to the other. MAGIC is hierarchical in nature and extracts each cell and then generates the necessary connect records to connect the nodes that are shared by more than one cell. Both of these approaches generate a simfile that has a fair amount of locality. Therefore the data structures must support a method of access that approximates the locality of reference inherent to the simfile.

These problems are all solved by the use of workspaces. Figure 5.5 displays the structure of workspaces. There are two system workspaces: the 'nodews' and the 'transws'. The nodews is a list of NODESTRUCTs and transws is a list of TRANSSTRUCTs. The memory is allocated for the NODESTRUCTs and the TRANSSTRUCTs as they are encountered in netlist description. This allocation scheme allows the locality of the netlist description to be inherited by the data space. Because the NODESTRUCTs and TRANSSTRUCTs are added to the system workspaces as they are allocated, the locality is also captured by the nodews and the transws. Therefore, a linear access to the workspace will approximate a linear access to the netlist description and will take full advantage of any locality that is present in the netlist description.

To create a user workspace the BooleanExpression† from the 'for' command is evaluated on each element in the source workspace. When the expression evaluates to TRUE the structure is added to the user workspace. This scheme maintains reference locality in the user workspace.

† Appendix B describes the grammar for a BooleanExpression in detail.

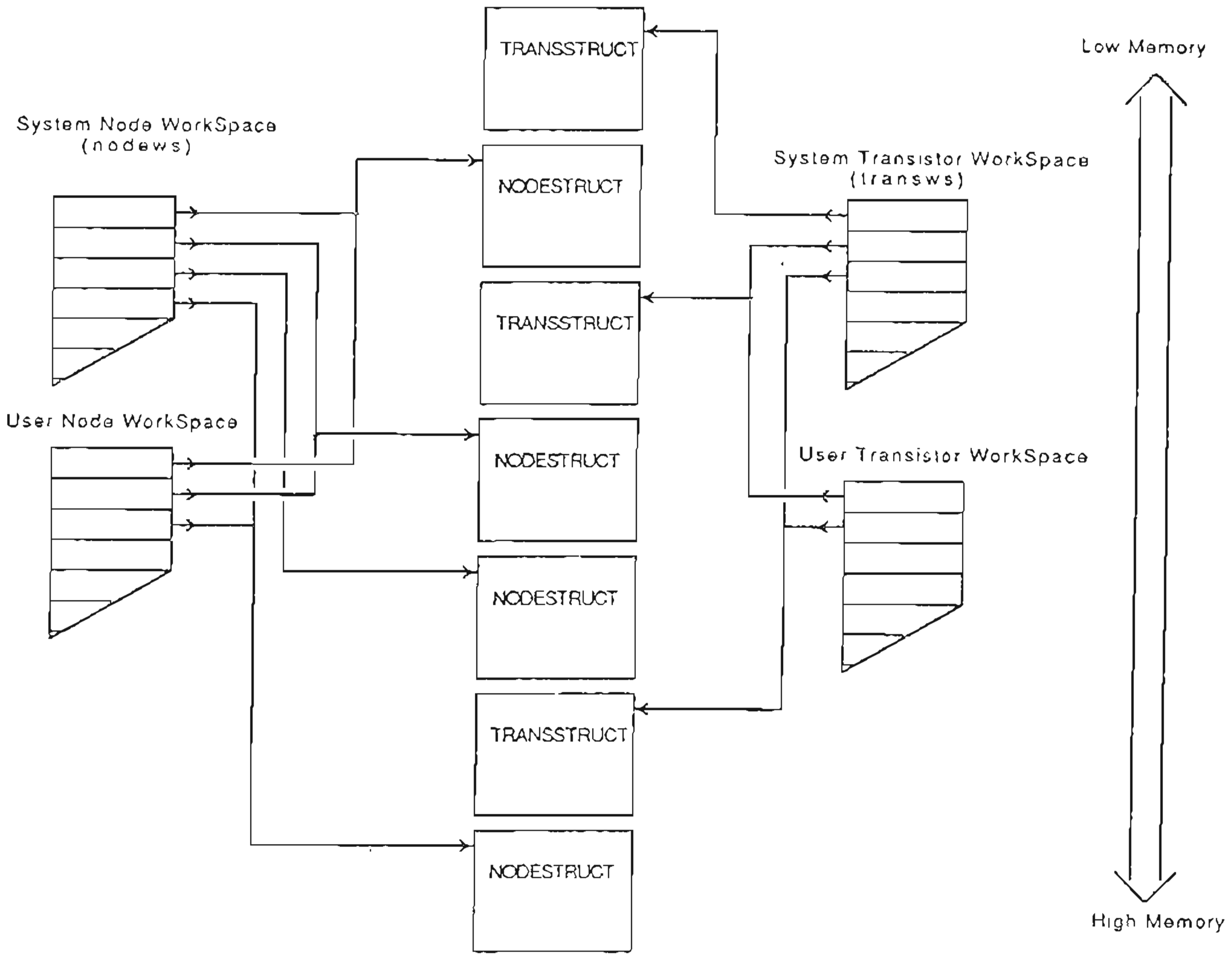


Figure 5.5: Structure of WorkSpaces

To summarize, the data structures behind the workspace provide five advantages:

1) *Data locality*. This approach makes every effort to take full advantage of any locality of reference present in the netlist description.

2) *Simple structure*. Because the workspace is a list, it is easy to create an efficient method to traverse every element in the list.

3) *Consistent Interface*. The access to a workspace is consistent whether it is a system workspace or a user workspace.

4) *Efficient memory usage*. All user workspaces are created dynamically as they are needed. In addition, all workspaces can dynamically 'resize' themselves if more space is needed.

5) *Each Entry in Workspace Unique.* Each entry in the workspace represents a unique node or transistor. This eliminates the need to make an additional pass to remove multiple references due to aliases.

5.1.2 Alias Data Structures

The implementation of aliases necessitates the implementation of two procedures. The first procedure can be used to 'connect' several existing nodes together. This procedure is called when 'connect' commands are encountered in the aliasfile and can only be called while the aliasfile is being parsed. This procedure can only be called at this time because it must 'collapse' several existing nodes into one node, which is a major modification of the data structures. The second procedure, which is much easier to implement, is used to create a new alias to an existing node. This procedure can be called by the user at any time with the 'setalias' command.

Figure 5.6 shows some of the data structures associated with four distinct nodes: 'A1', 'A2', 'P1' and 'A3'. Figure 5.7 shows how these data structures were changed and the new data structures that were created as a result of collapsing nodes 'A1', 'A2' and 'A3' into node 'P1'. Nodes 'A1', 'A2' and 'A3' are then aliases to 'P1' which is the primary node.

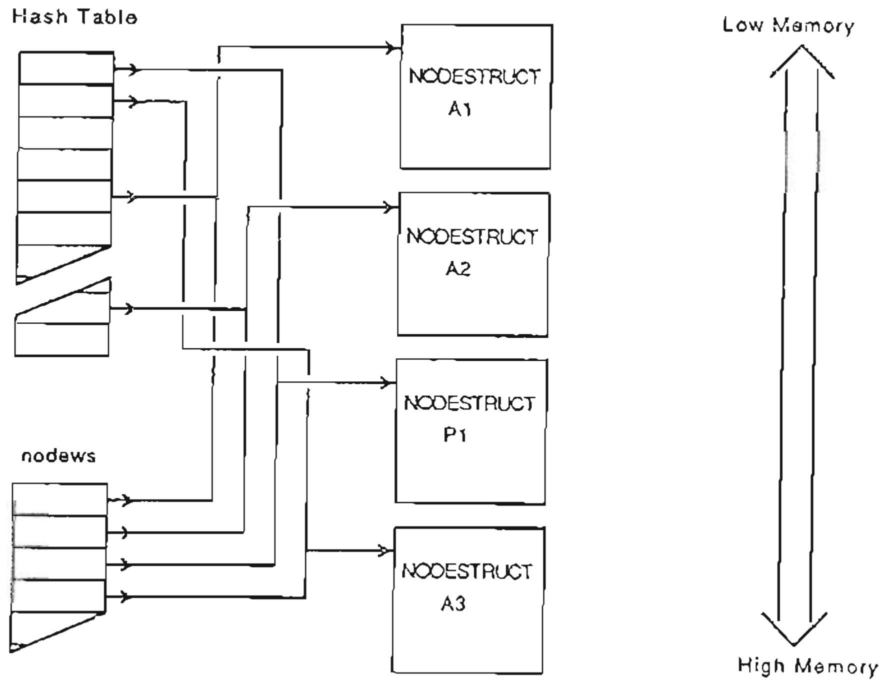


Figure 5.6: Data Structures for Four Distinct Nodes

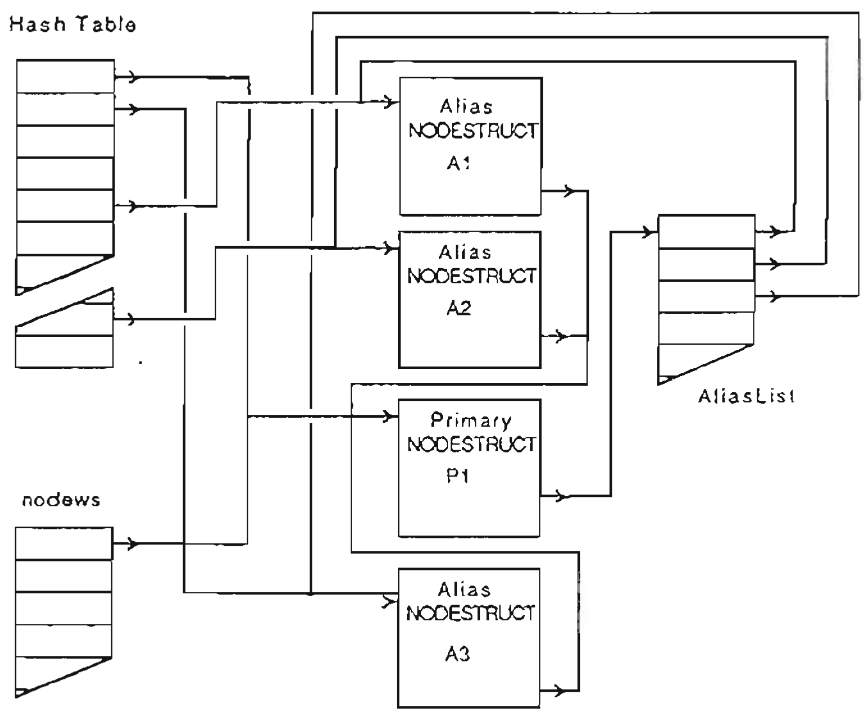


Figure 5.7: Data Structures for Three Nodes Aliased to the Fourth

To collapse nodes ‘A1’, ‘A2’ and ‘A3’ into node ‘P1’ the following operations must be performed:

1) If the NODESTRUCT for ‘P1’ is not already a primary NODESTRUCT, a list of NODESTRUCTs is allocated (AliasList). If NODESTRUCT for ‘P1’ is already a primary NODESTRUCT, then a larger list may have to be allocated. This list will point to all the NODESTRUCTs that contain aliases to node ‘P1’. The aliaslist allows direct access to all the node names that are aliases to ‘P1’. If ‘P1’ is already an alias then its primary node will become the primary node for node ‘A1’, ‘A2’ and ‘A3’. Figure 5.7 assumes that ‘P1’ is neither a primary node or an alias previously.

2) NODESTRUCTs for ‘A1’, ‘A2’ and ‘A3’ are marked as aliases.

3) The NODESTRUCTs for ‘A1’, ‘A2’ and ‘A3’ must contain a pointer to the primary node (‘P1’). This pointer allows direct access to the primary NODESTRUCT directly from any of the alias NODESTRUCTs.

4) The AreaCap (area capacitance) and the GateCap (gate capacitance) of all the alias NODESTRUCTs must be added to AreaCap and GateCap of the primary NODESTRUCT.

5) All pointers to lists of TRANSSTRUCTs (GateList, SouceList, DrainList) from the alias NODESTRUCTs must be added to the end of the corresponding list of TRANSSTRUCTs in the primary nodes (Figure 5.2).

6) All TRANSSTRUCTs that have pointers to the NODESTRUCTs (GatePtr, SourcePtr, DrainPtr) for nodes 'A1', 'A2' or 'A3' must now refer to the primary NODESTRUCT for 'P1' (Figure 5.3).
7) If nodes 'A1', 'A2' or 'A3' were previously primary NODESTRUCTs, then steps 5 and 6 must be performed on their alias NODESTRUCTs.
8) If nodes 'A1', 'A2' or 'A3' were already aliases to another primary node, then steps 2 through 6 must be performed on that primary NODESTRUCT and steps 5 and 6 must be performed on all its alias NODESTRUCTs.
9) The entries in the nodews that point to the alias NODESTRUCTs must be set to NULL. The primary NODESTRUCT will be the only entry to remain in the nodews.
10) The nodews must be compressed to remove all NULL pointers. Since all this is performed before user workspaces are created nothing has to be done to any of the user workspaces.

All the operations needed to create the data structures for the workspaces or the aliases can be time consuming; however, these operations only need to be performed once. What is more important is the speedy and orderly access to the necessary TRANSSTRUCTs and NODESTRUCTs by the interactive commands made possible by the data structures used to implement workspaces and aliases.

Since all the NODESTRUCTs in the nodews (and subsequently all user workspaces) are primary NODESTRUCTs, no checks are necessary to make sure a NODESTRUCT is a primary node. When an expression in a 'for' command contains a 'NodeName'† the interpreter has direct access to all the aliases through the AliasList.

In addition to being faster than using the hash table, this implementation allows for postfix name expansion when a '*' is the last character of the string. The implementation of workspaces also efficiently assures that all entries in the user workspace are unique. Workspaces also present a consistent interface whether or not it is a system workspace or a user workspace.

5.2 Memory Allocation

MOSSTAT does its own memory allocation for two primary reasons: speed and decreased internal fragmentation. malloc(), the memory allocator supplied with 4.2bsd UNIX is touted as being "a very fast storage allocator". But malloc() is a general purpose memory allocator that contains a great deal of functionality not needed in MOSSTAT. The following is a list of a few of those functions:

† Appendix B describes the grammar for NodeName.

1) malloc() attaches eight bytes of overhead that is attached to each allocation. This overhead is needed to capture free memory after it has been allocated and is not longer needed.

2) Memory is allocated from buckets that are at least twice the size of the allocation request. The minimum bucket size is 2 KBytes, and, therefore, requests for less than 1 KBytes will come from a 2 KByte bucket. Multiple requests of the same size will come from the same bucket. When the bucket is empty, another of the same size is requested. Allocation of memory from these buckets allows malloc() to speed up multiple requests that are of the same or similar size.

3) malloc() has an option that allows boundary checking to be done on memory allocated.

MOSSTAT has some specific memory allocation requirements and therefore its memory allocator is tailored to these specific requirements.

Memory is allocated by MOSSTAT for two different needs: workspaces and structures. The workspaces are large chunks of memory (minimum 8k) and they are only allocated a few times, typically less than 20. Workspaces are large and they will always be of a multiple of the page size. The space for the TRANSSTRUCTs, NODESTRUCTs and the GATESTRUCTs will be allocated in small blocks (less than 200 bytes) but there can be a large number of blocks (typically > 20,000).

The memory allocator in MOSSTAT is specifically designed for the allocation needs of MOSSTAT. Allocate() contains no support to allow the freeing up of memory that is no longer needed. MOSSTAT does not create any temporary data structures that could be re-used.

If the request is for more than the minimum size of a workspace (8 KBytes), then a system call is made to directly allocate the needed memory. Since the workspace sizes are a multiple of the page size and the allocation is done on a page boundary, the internal fragmentation is zero. Buckets are not used to reduce the number of system calls because it is known that the number of requests for more than 8 KBytes will be small.

If the request is for less than the minimum size of a workspace then the memory is allocated from a 32 KByte buffer that has been allocated on a page boundary. The large size of this buffer minimizes the internal fragmentation within the buffer. This methodology is consistent with the fact that there will be a large number of requests for these small blocks of memory. Small requests for memory (less than 1 KByte) to malloc() would be allocated out of a 2 KByte buffer (bucket). This bucket size would result in sixteen times as many system calls and increased internal fragmentation.

Allocate() does not require the computational overhead of multiple bucket sizes, or the space for the 8 Bytes per allocation needed to allow the freeing of memory. This reduced functionality makes Allocate() slightly faster and considerably simpler than malloc().

The end result is an application specific memory allocator that is much simpler that malloc(), is slightly faster, uses far fewer system calls and will have considerably less internal memory fragmentation.

5.3 Classify Gates Command

The 'classifygates' command will search for nodes that can be classified as 'logicgates'. A logicgate is the output node of a functional block that implements a boolean function. This functional block is a set of transistors and nodes, and may be a simple inverter or a complex nand-nor function. MOSSTAT may overlook some logicgates due to the limitations of the algorithm that it uses to locate logicgates. It is unlikely, however, that MOSSTAT will incorrectly identify a node as a logicgate when it is not.

The classification of the logicgate is a two step process. The first step is to identify the nodes that are logicgates and the second step is the actual classification of logicgates. The identification of the logicgates has four distinct steps:

1) The first step is to identify the power network.
2) The second step is to identify known logicgates.
3) The third step is to specify the direction in the power network.
4) The fourth step is to calculate the minimum and maximum resistance paths to VDD and GND.

5.3.1 Identify Power Network

The power net is the network of nodes that are in the paths from a logicgate to VDD and GND. The power net is made up of the vddnet, which contains all the nodes in the paths from a logicgate to VDD and the gndnet, which contains is all the nodes in the paths from a logicgate to GND. The vddnet is identified by propagating out from VDD through all transistors until an a node is encountered that is connected to the source or drain of an e-transistor. This node is the edge node of the vddnet and it is marked as being an invalid logicgate. At this time a structure is allocated to hold any parameters that are to be associated with the logicgates. The gndnet is identified by propagating out from GND through all transistors until a node is encountered that is connected to the source or drain of a p-transistor or until a node is encountered that is in the vddnet. This node is the edge node of the gndnet and it is marked as being an invalid logicgate. If a gate structure has not already been allocated for this node, one is allocated at this time.

The schematic in Figure 5.8 is an example of a typical CMOS complimentary logic gate. Nodes 'A' and 'Out' are members of a vddnet and therefore their 'VddNet' parameters are set TRUE. Nodes 'B' and 'Out' are members of a gndnet and therefore their 'GndNet' parameters are set TRUE. The vddnet starts at VDD and propagates out until 'Out' is encountered since 'Out' is connected to an e-transistor. The gndnet starts at GND and propagates out until 'Out' is encountered, since 'Out' is in a vddnet. 'Out' is now marked as an invalid logicgate.

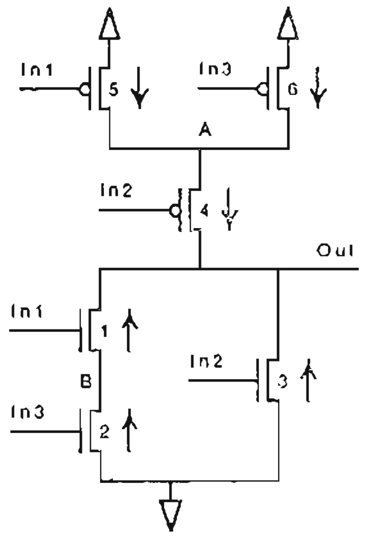


Figure 5.8: CMOS Logic Gate

5.3.2 Specify Direction in Power Network

As the nodes are added to the power network, the 'direction' is specified at each transistor in the power net. In the vddnet the direction goes from VDD along each path to the logicgate. In the gndnet the direction goes from GND along each path to the logicgate. A path is a unique set of transistors in series that form a path to GND or VDD. Put another way MOSSTAT assumes that two parallel transistors are never enabled at the same time. Any transistors that are in parallel are considered to be part of two distinct paths to GND or VDD. This is a worst case assumption and can only be made more accurate by using input vector information.

The specification of the direction in the power network is done as the nodes are added to the power network. When a node is added to the gndnet all the transistors that are connected to that node are checked. If that node is 'A' and 'A' is connected to the source of a transistor whose drain is already in the gndnet then the direction in that transistor is set to be from the drain to source. This is possible because any node that has already been added to the gndnet is 'upstream' (closer to GND) from the node that is just being added to the gndnet. The same process is repeated for any transistors whose drain is connected to 'A'. The specification of the the direction in the vddnet is identical for the vddnet.

Once the direction has been specified for each transistor in the gndnet and the vddnet some data structures are created for quick access to each transistor in each path to VDD and GND. The task of identifying the paths to VDD and GND is quite easy because the direction at each transistor has already been specified. Figure 5.9 shows the data structures that would be allocated for the logicgate in Figure 5.8. The TRANSSTRUCTs in 5.9 have been numbered in the same way as the transistors in Figure 5.8. In Figure 5.8 there are two paths from VDD to 'Out': 'VDD'->'A'->'Out' and 'VDD'->'A'->'Out'. There are also two paths from GND to 'Out': 'GND'->'B'->'Out' and 'GND'->'Out'.

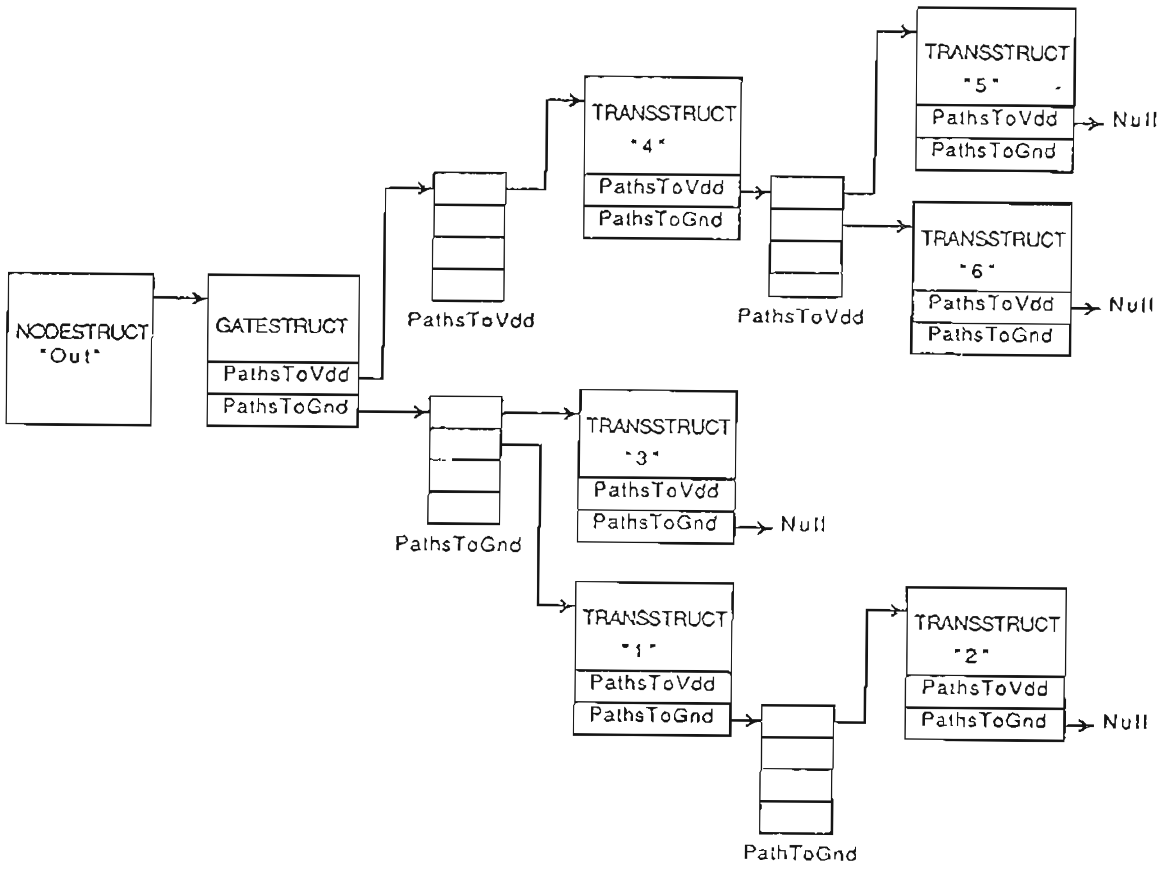


Figure 5.9: Logic Gate Data Structures

5.3.3 Identify Known Logic Gates

The node workspace (nodews) is then searched for invalid logicgates. Any invalid logicgate that is a member of both the vddnet and the gndnet is now marked as being a valid logicgate. Node 'Out' in Figure 5.8 would be marked as a valid logicgate. All nodes marked as invalid logicgates which are not in both the vddnet and the gndnet are left marked as unknown logicgates. Both the unknown and known logicgates are added to the logicgate workspace (gatews). Any node which has been classified as a logicgate is a member of two system workspaces: nodews and gatews.

5.3.4 Calculate Minimum and Maximum Resistance Paths

Now that we have identified all the paths from VDD and GND, we can calculate the resistance of each path to VDD and GND. The maximum resistance to GND, minimum resistance to GND, maximum resistance to VDD, and minimum resistance to VDD are all identified and stored in the 'MaxRtoGnd', 'MinRtoGnd', 'MaxRtoVdd' and 'MinRtoVdd' parameters of the GATESTRUCT. In Figure 5.8 if we assume that all the transistors have one square of resistance, the MinRtoVdd, MaxRtoVdd would both be equal to two squares. In the gndnet the MinRtoGnd will be one square and the MaxRtoGnd will be two squares.

5.3.5 Classify Gates

Now that all the logicgates have been identified the next step is to do the actual classification of the gates. All logicgates will be classified into one of the following groups:

Dynamic.
If all of the paths from VDD and GND to the logicgate are 'clocked' paths, then the logicgate is classified as dynamic. A clocked path is a path where at least one of the transistors in the path is gated by a clocked node. A clocked node is a node which has been declared as a clock or is classified as a valid dynamic logicgate. The logicgate in Figure 5.10 will be classified as dynamic.

Invalid Dynamic.
If some but not all of the paths from VDD and GND to the logicgate are clocked, then the logicgate is classified as an invalid dynamic gate.

Pseudo NMOS.
If the logicgate has a single path to VDD and that path is through a single p-transistor and that p-transistor is gated by GND, then the logicgate is classified as a pseudo NMOS gate.

Invalid Pseudo NMOS.
If the logicgate has more than one path to VDD and at least one of those paths is a single p-transistor that is gated by GND, then the logic is classified as an invalid pseudo NMOS gate.

NMOS.
If a logicgate has a single path to VDD and it is through a single e-transistor or single d-transistor then it is classified as an NMOS Gate.

Invalid NMOS.
If a logic gate has a more than one path to VDD and it is through a single e-transistor or d-transistor this it is classified as an Invalid NMOS Gate.

Static CMOS.
If a logicgate does not fall into any of the above classifications then it is classified as an Static CMOS logicgate. Since this classification includes all logic gates not classified into other groups it may contain both valid and invalid logic gates. The logicgate in Figure 5.8 will be classified as static CMOS.

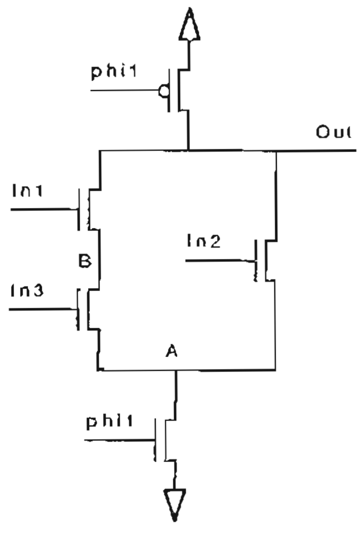


Figure 5.10: Dynamic Logic Gate

If the logicgate is classified as dynamic then MOSSTAT will calculate the 'net capacitance'. The net capacitance is the total capacitance of all the nodes in the gndnet and the vddnet with the exception of VDD, GND and the logicgate itself. The net capacitance of the dynamic gate (Figure 5.10) would be sum of the area capacitance and the gate capacitance of node 'A' and node 'B'. The result is stored in the 'NetCap' parameter of the GATESTRUCT (Figure 5.4). The 'NetCap' parameter can be used to enforce a portion of the Clocking Rules for Dynamic MOS Systems [Bra86]. These Dynamic Clocking Rules are briefly described in the glossary in Appendix C. In practice the net capacitance will often be zero, because the nodes in the power network are not connected to any transistor gates and will generally have negligible area capacitance. MOSSTAT assumes that the minimum net capacitance is .1 femtofarads.

The net capacitance is the total amount of capacitance that is contained in all the 'internal' nodes of gate. If the inputs change value while the clock is high there is now drive on the output and internal charge sharing could possibly change the value of the output. A portion of the clocking rules assures that there is adequate charge on the outputs to ensure that the state of the output will not change due to the charge sharing caused by the inputs changing state while the clock is low. If the ratio of the capacitance on the outputs to the net capacitance is greater than five then the value of the outputs cannot change due to charge sharing in the net caused by inputs changing value while the clock is low. Since MOSSTAT assumes that the net capacitance is at least .1 femtofarads, the logicgate must have at least .5 femtofarads of total capacitance. In practice this means that the logicgate must be connected to the gate of a transistor in order to have the required capacitance. It would be unusual for any nodes in the GndNet or the VddNet to be connected to transistor gates.

The 'for' command can be used to create a workspace that contains the logicgates that have a capacitance/NetCap ratio in any range specified by the user. The syntax for the 'for' statement is discussed in Section 4.2 of Appendix A.

6. Evaluation of Mosstat as a Design Aid

The evaluation of MOSSTAT as a design aid has been done with two different designs. The first design, the PRSC, is a small CMOS design implemented by the Fall 1985 VLSI design class at OGC as a lab assignment. The second design, the Object Map Chip is a large NMOS design that was partially implemented by the author at Tektronix Laboratories.

6.1 PRSC

MOSSTAT was used by the Fall 1985 VLSI design class at OGC as part of a lab assignment. The assignment was to lay out a small CMOS design using MAGIC. This design implemented a pseudo random sequence counter (PRSC). The design was specified for the students with a plot of the physical layout and contained 125 transistors and 66 nodes.

The class consisted of students with a wide diversity of backgrounds. Some had software backgrounds, others had hardware backgrounds, and most had a combination of both.

A portion of the assignment called for the students to use MOSSTAT to find the critical timing path in the design. Critical time path identification is only one of many possible applications for MOSSTAT. It was however the only application that was required.

6.1.1 User Feedback

In general, most students found MOSSTAT useful in isolating some types of problems. They reported that the complexity of the tool minimized its usefulness on a design as simple as the PRSC . The documentation was rated good, but needed to be expanded, especially in the number of examples given. The PRSC, however, was the first CMOS design to be analyzed by MOSSTAT. These students were also the first real users and they found numerous bugs.

Some students were asked to list the rule check commands according to their relative usefulness in error detection and isolation on the PRSC. The results are displayed in Figure 6.1.

It is interesting to note that the users rated the usefulness of a command to be the inverse of the complexity of the command. Considering the time constraints on the students for the lab and the simplicity of the design, it is not surprise to find that the tended to use less complex analysis methods.

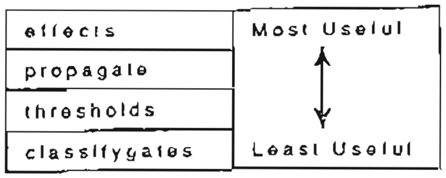


Figure 6.1: Command Usefulness

6.1.2 Performance

It took approximately 2 seconds for MOSSTAT to read in the netlist description and to create the internal data base. After that point all interactive commands ran instantaneously (less than 1 second).

6.1.3 Changes Made to MOSSTAT

Some new commands were added at the request of users. They were 'propagate', 'reset' and 'setalias'.

Several complaints were made against the use of spaces to delimit keywords in the interactive command language. Users also complained about the use of upper case characters in keywords. The scanner for MOSSTAT now ignores the case of the characters in keywords. Case sensitivity is retained for node names. Spaces between keywords is now optional except where spaces are necessary to separate node names. Also, a second version of the manual was created that contains more examples.

MOSSTAT has now been more thoroughly debugged and more extensively documented. What is needed is a chance to see MOSSTAT used on larger student projects.

6.2 Object Map Chip

The Object Map Chip implements the Object Map memory, the memory searching, address range checking and absolute address calculation for the Magnolia Object Bus Interface. The design was to have been implemented in NMOS but the physical design was never completed. MOSSTAT was used to evaluate the netlist description of the Object Map Chip that was generated from a schematic description of the circuit. This netlist description contained 7898 transistors and 3325 nodes, and previously had been thoroughly simulated and analyzed with RNL and STAT.

6.2.1 User Feedback

Due to the complexity of the design, MOSSTAT found a large number (several hundred) of discrepancies in the netlist description. As it turned out, most of these discrepancies were due to the language that is used to generate the netlist description. These discrepancies necessitated a great deal of interaction with the database in order to investigate the context of the discrepancies and verify the validity of that portion of the design.

Most of the discrepancies were due to numerous nodes that were either not effected by an input or could not effect an output. Most of these in turn were due to extraneous circuitry or gates connected to VDD or GND. In all cases except one, the discrepancies were due to the way the netlist description was generated. The hierarchical schematic description language made it difficult to avoid creating unneeded circuitry such as transistors gated by VDD or GND. Since extra circuitry does not effect the functionality of the design, they can be ignored. With the 'for' command it was easy to isolate the source of these errors by looking at the nearby transistors and nodes.

When STAT was used to analyze the Object Map Chip it generated over 4500 lines of error statements. Most of these error statements were due to the cascading of errors. The batch nature of STAT made it necessary to read and evaluate all of these error statements. The analysis of these error statements led to the discovery of most all of the discrepancies uncovered with MOSSTAT. Incidently the Object Map Chip is a circuit that is almost completely debugged, the number of error statements that had been generated by STAT in the early stages of design were several times this amount.

MOSSTAT found that it was possible to set all nodes both high and low, which is not surprising considering the amount of prior simulation. One error which was discovered with MOSSTAT and which was overlooked by previous tools was that there were many logic gates that were classified as NMOS Gates that had multiple paths to VDD. This was caused by a faulty net description that specified the same transistor more than once. Two transistors are the same if they have the same gate, source and drain. What is interesting to note is that this error was easily discovered with MOSSTAT, but was not uncovered through extensive simulation, because it did not effect the functionality of the design.

6.2.2 Performance

It took approximately 4 minutes for STAT to read in the netlist description, create the internal database and do all its analysis on a lightly loaded VAX 780. The process size was 725 KBytes.

It took approximately 20 seconds for MOSSTAT to read in the netlist description and to create the internal data base. After that point the 'for' command typically took 1-2 seconds and the worst of the rule checks (classifygates) took approximately 15 seconds. These performance figures are for a single user workstation with an adequate amount of real memory, so that paging activity were kept to a minimum. The process size was 1008 KBytes.

6.2.3 Changes Made to MOSSTAT

The complete analysis of the Object Map Chip with MOSSTAT came late in the testing phase of MOSSTAT, and consequently few changes were made as a direct result of the analysis done on the Object Map Chip. The ability to list the inputs, outputs and clocks without the use of the 'for' command was the only visible change made at this time. In the early stages of development of MOSSTAT, the Object Map Chip was used extensively to test the performance of MOSSTAT, and, consequently, many of the performance enhancements discussed in Section 5 are a result of the analysis done on the Object Map Chip.

6.3 Conclusion

MOSSTAT does much to validate the viability of the interactive approach to static analysis of MOS designs. MOSSTAT does many of the same types of analysis as other static analysis tools (STAT and ERC), and, consequently, will detect the same types of errors as those tools. The interactive nature of MOSSTAT makes it much more effective than previous static analysis tools at isolating errors.

The run time for MOSSTAT analysis is considerably less than either STAT or RNL. It took less than one minute to run MOSSTAT versus over four minutes to run STAT and over five minutes for PRESIM (the tool that creates the binary netlist description used by RNL).

Owing to the need for a simulator such as RNL to propagate errors to node whose values are being monitored, and the powerful query commands available in MOSSTAT, MOSSTAT is more effective at error detection and isolation than such a simulator. MOSSTAT was also able to detect some errors in the Object Map Chip that were not detected by even many months of simulation efforts by the author.

MOSSTAT is not simple tool to use, and its usefulness is questionable for small designs. In any design, simulation is an excellent method of detecting errors. In a small design of low complexity such as the PRSC, error isolation using simulations is not in general much of a problem. In larger designs error isolation can be difficult and simulators are of limited usefulness for error isolation.

MOSSTAT falls short however in a couple of areas. The execution time for some of the 'for' commands is slower than the performance goals laid out in the specifications. Their performance, however, still does not interfere with the interactive nature of MOSSTAT. Another detraction was the complexity of MOSSTAT. This problem is particularly evident in a simple design such as the PRSC. There are number of areas where the functionality of MOSSTAT could be easily increased, thus enhancing its usefulness; these areas will be discussed in Section 7.

7. Conclusions and Future Work

The function of a static analysis tool for MOS digital designs has been described. Performance goals that must be met to insure the productivity of the tool have been specified. The algorithms and data structures to implement this tool have been described.

The analysis approach that this tool utilizes is compared with the approaches of other tools. A design methodology for digital MOS designs is specified and the role the tool plays in this methodology is described.

Two different designs have been analyzed with this tool, and one of the designs was analyzed by users other than the author and their feedback was evaluated. One of these designs was implemented in CMOS, the other was in NMOS. The tool is shown to behave as specified.

What is yet needed to prove that the tool as specified can play an important role in the design methodology for digital VLSI circuits. This can only be done by giving a number of designers access to MOSSTAT and waiting for their evaluations.

There are a number of features and fixes to MOSSTAT that would greatly enhance its usability. Section 7.1 will discuss these features.

MOSSTAT attempts to shed some new light on a number of different approaches to be taken to detect and isolate errors in a circuit design. There were a number of issues however, that were not addressed at this time. These issues are an extremely important part of the 'ideal' design environment that was alluded to in Section two. Section 7.2 will list some of the new analysis approaches that could be pursued.

7.1 Additional Work on MOSSTAT

There are several ways that the existing version of MOSSTAT could be enhanced. These changes fall into four categories: increasing functionality, portability, performance and bug fixes.

7.1.1 Functionality

MOSSTAT could be made a much more effective tool for critical timing path identification. Rise times and fall times can currently be calculated at a node that has been classified as a logic gate. Presently MOSSTAT assumes that any pass transistors (or transmission gates) that are encountered are turned off. What is needed is the ability for the user to specify if a given pass transistor is normally open or closed. With this knowledge MOSSTAT could be enhanced to include the capacitance of nodes normally driven by a logic gate in its rise time and fall time calculations. The most obvious approach would be to assume that all the pass transistors are normally open and then allow the user to over ride the default where appropriate. The interactive nature of MOSSTAT should make this feature easy to implement.

7.1.2 Portability

The scanner for the interactive command language is generated with LEX. This was the appropriate choice during the development phase of MOSSTAT. The problem is that the scanner generated by LEX is quite large (8000 lines) and consequently the scanner that it generates needs a process stack of over 1 MByte which is larger than most UNIX shells allow. The scanner generated is large for two reasons. The first is that the language allows keywords to by entered in upper or lower case letters. This causes LEX to generate multiple states for each keyword. The second reason is that the scanner had to hold 'state' information to allow for context dependent scanning. This state information is needed because a valid node name can contain characters that are also valid as tokens. In order to eliminate the need for white space to separate tokens the scanner had to have the ability to reject some tokens in certain situations. The language is fairly stable and it would now be appropriate to write a scanner rather than use LEX to generate one.

7.1.3 Performance

Considerable effort was put into performance 'profiling' in the early stages of MOSSTAT's development. A large number of changes have been made to MOSSTAT since that time and it is likely that some additional 'profiling' would point out a number of performance bottlenecks. Some of these bottlenecks could be either eliminated or minimized with a few minor changes to the existing code.

NODESTRUCTs and TRANSSTRUCTs are allocated from the same buffer space. If the NODESTRUCTs and TRANSSTRUCTs were allocated from different buffers then the page fault rate could be reduced for most types of access to the nodews (any access that does not reference a TRANSSTRUCT). This allocation scheme would not increase the page fault rate for other types of access to either the nodews or the transws.

MOSSTAT allocates a full NODESTRUCT for an alias even though the space is largely unused. Allocating this extra space is done because in many cases this space was already allocated for the node before it became an alias. Freeing up the space would do little good unless some form of garbage collection were performed. It would, of course, depend on the circuit, but in one example there are 14,000 aliases for 3,000 nodes which would result in a savings of 1.8 MBytes of memory (1/3 of total memory requirement).

7.1.4 Bug Fixes

There are a number of known bugs that prevent MOSSTAT from performing as specified. Most of these bugs are due to a number of changes that have been made since the Fall 1985 VLSI Design class used MOSSTAT. Most of these bugs are in the scanner, which needs to re-written for the reasons alluded to above. There are also undoubtably a number of unknown bugs. The 'classifygates' command is particularly suspect, because of its complexity and because few users other than myself have carefully evaluated its results. The interpretor would also be suspect because of the number of branches occurring in the code.

7.1.5 More User Feedback

The biggest shortcoming of this whole project is the lack of input from users. MOSSTAT implements a variety of interesting algorithms but the real question that is yet to be answered: Is MOSSTAT usable for typical users? A great deal was learned from the feedback that was received from the students that used MOSSTAT. Before future static analysis tools are investigated, much more should be learned about the effectiveness of the rule checks and interactivity of MOSSTAT.

7.2 Future Static Analysis Tools

There are several approaches that could be incorporated into future static analysis tools but were not within the scope of the MOSSTAT project. The first one is to use 'rule based' analysis, the second is to perform power estimation and the third is the need to do away with extraction.

7.2.1 Rule Based Analysis

More work is necessary to apply static analysis techniques to the design of VLSI circuits. One of the most promising of these approaches is the encapsulation of a 'knowledge base' associated with design of circuits. This knowledge base must support incremental additions by the user. The tool must 'perform inferrences' about the design using rules contained in this knowledge base. The result will be a tool that can 'critique' a digital circuit design using knowledge captured in the knowledge base.

7.2.2 Power Estimation

Future static analysis tools need to help the user with power estimation. This analysis should include the analysis necessary to calculate the minimum acceptable widths of metal lines in the circuit[Wil86].

7.2.3 Extraction

A physical design must be extracted to create a flat netlist description that can be used by analysis tools and simulators. Most tools need 5-30 minutes to convert this netlist description into an internal representation of a large circuit. What is needed is the incremental extraction of this internal representation. Incremental extraction would eliminate the need for batch extraction of the circuit design.

The elimination of batch extraction is the single biggest difference between the existing design environment and the 'ideal' environment. This would call for considerable integration of tools at all levels from user interface down to data representation [UW85A].

References

[Bak80] Clark Baker, "Static Analysis Program", MIT Laboratory for Computer Science, Cambridge, Ma., 1980.

[Bra86] A. C. (Kit) Bradley, "Some Simple Clocking Rules for Dynamic MOS Systems", Technical Report No. CRL-86-21, Computer Research Lab, Tektronix Laboratories, Tektronix, Inc., 1986.

[Bry83] Randal E. Bryant, "A Switch Level Model and Simulator for MOS Digital Systems", Computer Science Department, California Institute of Technology, Pasadena, Ca., 1983.

[Cry83] J. Ousterhout, "Crystal: A Timing Analyzer for nMOS VLSI CIrcuits", Proceedings of the Third Caltech VLSI Conference, 1983.

[Eps77] R. Epstein, "A Tutorial on Ingres", Memo UCB/ERL M77-25 Dept. EECS, University of California, Berkeley, 1977.

[Joh86] Timothy Johnson, "Mosstat: An Interactive Static Rule Checker for MOS VLSI Designs, Source Code", Oregon Graduate Center, Beaverton, OR, April, 1986.

[Kel84] Van E. Kelly, "The CRITTER System Automated Critiquing of Digital Circuit Designs", ACM IEEE Design Automation Conference, 1984.

[Lob84] C. Lob, "RUBICC, A Rule-Based Expert System for VLSI Integrated Circuit Critique", Memo UCB/ERL M484/80 Dept. EECS, University of California, Berkeley, Sept. 1984.

[Mea80] C. Mead and L. Conway, "Introduction to VLSI Systems", Addison-Wesley, Mass., 1980.

[Noi82] Noice, D., Mathews, R., and Newkirk, J., "A clocking descipline for Two-Phase Digital Systems", Proc. ICCC 82, IEEE, September, 1982.

[Ost83] J. Ousterhout, G. Hamachi, R. N. Mayo, W. S. Scott and G. S. Taylor, "Magic, a VLSI Layout System", Proceedings of the 21st Design Automation Conference, 1984, 542-548.

[Pen82] Keith Pennick, ERC Man Page, "VLSI Design Tools, Reference Manual", UW/NW VLSI Consortium, Department of Computer Science, University of Washington, Seattle, Wa., 1984.

[Spi85] A. Vladimirescu, Kaihe Zhang, A. R.Newton, D. O.Pederson, A. Sangiovanni-Vincentelli, "Spice User's Guide", "VLSI Design Tools, Reference Manual", UN/NW VLSI Consortium, Department of Computer Science, University of Washington FR-35, Seattle, Wn., 1985.

[Ter83] Cristopher Termin, "Simulation Tools for Digital LSI Design", VLSI Memo No. 83-154, MIT Laboratory for Computer Science, Cambridge, Ma., 1983.

[TV83] N. P. Jouppi, "TV: An nMOS Timing Analyzer", Computer Systems Laboratory, Stanford University, 1983.

[UW85] "VLSI Design Tools, Reference Manual", UW/NW VLSI Consortium, Department of Computer Science, University of Washington FR-35, Seattle, Wa., 1985.

[UW85A] "VLSI Design Generators", (From slides of a presentation), UN/NW VLSI Consortium, Department of Computer Science, University of Washington FR-35, Seattle, Wa., 1985.

[Wil86] Jeffrey Wilson, "Analysis of Power Requirements inside of NMOS Integrated Circuits", Oregon Graduate Center, Beaverton, Or., 1986.

Appendix A: Mosstat User's Manual

1. Introduction

MOSSTAT is an interactive static rule checker for NMOS and CMOS VLSI circuit designs. MOSSTAT allows the user to interactively select from a number of different static rule checks to verify a design. MOSSTAT initially creates as 'database' that will contain all the information retrieved from the netlist description of the circuit. The result of the rule checks will also be stored in this 'database'. MOSSTAT provides the user with a rich set of commands for retrieving information contained in the 'database' such as connectivity, transistor sizes or node capacitance.

MOSSTAT is most helpful when used in the early stages of verification after the circuit has been designed and laid out either schematically or physically. Although most of the violations found with MOSSTAT can be found with a dynamic simulator, MOSSTAT will provide a more productive environment for the detection and isolation of these errors.

MOSSTAT is written in 'C' and although it was originally developed in a UNIX environment it should port to any system with a complete 'C' compiler and the standard 'C' libraries. For more information about the porting of MOSSTAT to other environments see the file 'README' in the MOSSTAT source directory.

MOSSTAT will use from from 300 to 600 bytes per transistor, so the memory requirements can be quite large. This situation is made worse, because, although the analysis done by MOSSTAT has some data locality, the 'working set' of pages needed by MOSSTAT can still be quite large.

MOSSTAT is designed to be highly interactive on designs as large as ten to fifteen thousand transistors. The speed of MOSSTAT will degrade as the number of transistors increases. The performance degradation will be linear with respect to the number of transistors as long as paging is kept to a minimum. The large working set of pages needed requires that MOSSTAT and its data remain resident almost entirely in memory to avoid thrashing.

This manual is a tutorial on how to use MOSSTAT to execute rule checks and evaluate the results of these static rule checks. If is a detailed description, for a more concise description see the UNIX man page for MOSSTAT. As with most programs, MOSSTAT is easiest to understand when it is used on test cases as you read this document.

A short description of the theory behind each rule check is included in the section describing the Rule Check Commands. To obtain more detailed description of the algorithms used refer to: "Mosstat An Interactive Static Rule Checker for MOS VLSI Designs".

A glossary of some of the terms used in this manual is included. These terms will be used frequently with the assumption that the reader already has an understanding of their meaning.

2. Starting Mosstat

To run MOSSTAT type

mosstat [-c commandfile] basename[.sim]

It is assumed that 'basename.sim" is a valid simfile[UW85]; and MOSSTAT will make little effort to recover from errors detected in the simfile. Either the basename of the simfile or the entire name may be specified.

All output from MOSSTAT, by default, is sent to 'standard out' with the exception of error statements, which are sent to 'standard error'. The output of the 'print' command can optionally be directed to a file.

MOSSTAT starts by telling the user its version number. Next, MOSSTAT reads in the net description from the simfile. MOSSTAT will notify the user that it is opening the simfile (basename.sim) and the aliasfile (basename.al). As it reads in the net description, it creates the internal 'database' that contains all known information about the circuit. MOSSTAT then reads in the 'basename.al' file that is a list of all the node aliases.

Aliases may be used to connect nodes which are distinct from each other. MOSSTAT must then collapse all the occurrences of a node into one primary node that can have any number of aliases. The primary node 'inherits' all the connections from all its aliases. The node and gate capacitance of the primary node is the sum of the gate and node capacitance of all of its aliases. The following is an example of a valid alias record:

```
= nodename1 nodename2 nodename3 . . .
```

nodename1 will be the primary node name and all the other node names in the list will be aliased to the primary node. If nodename1 is already an alias to a primary node, then all the node names in the list and nodename1 will be aliased to that primary node. If any of the node names in the list other than nodename1 are already primary nodes then those primary nodes and all their aliases will be aliased to nodename1. In this fashion distinct nodes can be either connected together or we can create multiple node names that refer to the same node. An example of this practice are the following aliases that are by default created by MOSSTAT:

"Vdd" and "vdd" are aliased to "VDD".

"Gnd" and "gnd" are aliased to "GND".

These aliases are created after the aliasfile has been read and therefore VDD and GND will always be the primary node name for power and ground. The user can specify power and ground with any of the above aliases. When these aliases are created a check is made for the existence of power and ground. If power and ground have not been specified by the user then MOSSTAT will issue a warning message and exit.

Information about all the known aliases to a particular node can be retrieved with the 'alias' command. The 'alias' command will be discussed in Section 3 of this appendix.

Many users will be using MAGIC[Ost83] to specify the physical description of their circuit. If this is the case then the 'aliasfile' is extremely important because of the way that the extractor from MAGIC uses aliases. MAGIC's extractor is hierarchical and it uses aliases to connect nodes in order to flatten out the hierarchy in the circuit.

MOSSTAT next checks for the existence of a command-file specified with the -c option. The -c option specifies that the text in the command-file will be processed as if the commands had been entered interactively from the keyboard. The syntax of the command-file is exactly the same as the syntax specified in the MOSSTAT Interactive Environment, which is discussed in section 3.1. If the command-file does not contain a 'quit' command then MOSSTAT will return control to the terminal when it is finished processing all the commands in the command-file. If a 'quit' command is encountered then MOSSTAT will quit processing the command-file and exit immediately. More than one file can be loaded with this option by either using multiple '-c' options or including a 'source' command from within the command-file. The user must avoid cyclical reference within command-files.

MOSSTAT knows about three different transistor types. Depletion n-type transistors and enhancement n-type transistors which will hereafter be referred to as d-transistors and e-transistors respectively. The third type of transistor is enhancement p-type transistors which will hereafter be referred to as p-transistors.

As far as MOSSTAT is concerned the source and the drain of a transistor are totally interchangeable and there is electrically no difference between the two. For consistency MOSSTAT will retain the same source - drain orientation as is specified in the simfile. It is therefore perfectly acceptable to connect the source of transistor to VDD or connect the drain of a transistor to GND.

The creation of the internal 'database' can be time consuming in large designs due to the nature of the 'database'. The primary design consideration for the internal data structures is interactive access speed. To achieve the desired interactive performance the data structures tended to be more complex, larger and more expensive to create. Section 5 of "MOSSTAT An Interactive Static Rule Checker for MOS VLSI Design" discusses the data structures. The database for a circuit of about eight thousand transistors will take approximately twenty seconds to five minutes to create on a lightly loaded VAX 780 depending on the number of aliases. Once this 'database' has been created, MOSSTAT is ready to accept input from the user and will prompt the user with a '.'.

3. Mosstat Interactive Environment

After MOSSTAT has created an internal 'database' from the simfile and the aliasfile and read in all the command-files, it is ready to accept interactive commands from the user after the prompt '.'.

After the prompt the only way to exit from MOSSTAT with a normal exit status is by executing the 'quit' command. If any interactive commands are executing and the user types an interrupt character (usually a ^C) the command will be interrupted and MOSSTAT will return to the prompt. A quit character (usually a ^\ ) will cause MOSSTAT to terminate immediately. MOSSTAT does not attempt to catch the suspend character (usually a ^Z). If your shell supports job control you can suspend MOSSTAT and return to the shell.

All reserved words are in **bold** print. A newline or a ';' (semicolon) is used as a command terminator. A '\' before a newline or carriage return allows a command to be continued on the next line. All characters between a leading '{' and closing '}' are treated as comments and are discarded. The use of this feature in command files that are to be loaded with the 'source' command is encouraged.

You may notice that most commands have some of the trailing postfix characters enclosed in [ ]'s. All postfix characters enclosed in the [ ]'s are optional and do not need to be typed in. Some arguments in a command may be enclosed in [ ]'s, These arguments are optional and are only needed to override defaults.

In all documentation for clarity the key words are specified as all lower case characters and the parameters are specified as a combination of upper and lower case characters. Key words must be specified by the user as all lower case letters. Parameters may be specified identical to the documentation or as all lower case letters, whichever is easier for the user. A node name may contain most any printable character and is therefore case sensitive.

The scanner allows the user to use white space (tabs, or spaces) for token separators. In most situations the white space is optional. For a more complete discussion of his concept see the grammatical description of a BooleanExpression see Appendix B.

There are three groups of interactive commands. The *Query Commands* can be used to query the 'database', create 'user workspaces' or print out the contents of a workspace. The *Rule Checks*, perform a specific set of rule checks on a circuit and the results of these rule checks are written into the 'database'. The *Miscellaneous Commands* are all other commands that don't fit into the first two groups.

3.1 Query Commands

The Query Commands retrieve information from a workspace. A 'workspace' is a list of entries the 'database'. There are two different kinds of workspaces: a user workspace and system workspace. A system workspace is created once by MOSSTAT and cannot be changed by the user. There are three system workspaces:

nodews	The node workspace is a list a elements that have type NODE. Each element in the nodews contains all known information about a particular node in the circuit.
transws	The transistor workspace is a list of elements that have type TRANS. Each element in the transws contains all known information about a particular transistor in the circuit.
gatews	The gate workspace is a list of elements that have type GATE. Each element in the gatews contains all known information about a particular logicgate in the circuit. Any element which is a logicgate is also a node, so a logicgate is a member of both the gatews and the nodews.

A user workspace is a temporary list which can created and re-used many times by the user. There are eight user workspaces: w0, w1, w2, w3, w4, w5, w6 and w7. The memory for the workspaces is dynamically allocated as it is needed. The allocation of memory for the workspaces is done without intervention from the user.

There are three Query Commands: 'count', 'for' and 'print'. These commands enable the user to classify elements in the database, to retrieve information from the data base, and to create workspaces that contain the elements specified by the user.

The 'count' command can be used to get a quick overview of the contents of a workspace. The 'for' command is used to create a user workspace from either another user workspace or a system workspace. The expression syntax of a 'for' command allows the user to specify the contents of the workspace. The 'print' command is used to print the contents of a workspace.

3.1.1 Count Command

count *workspace*

The output of the 'count' command is dependent upon the type of the workspace.

If the workspace has a type of NODE, then the type of the workspace, the number of nodes and the number of nodes that have aliases is given.

If the workspace is of type TRANS, then a report is generated showing the number of each kind of transistor. The transistors are first partitioned according to their type. Then each each type of transistor is classified according to other characteristics. The types supported and the classes associated with each type are listed below:

e-transistors		
	funny	(The gate and the drain **or** the gate and the source **or** the source and the drain are the same)
	fixed gate	(The gate is connected to VDD or GND)
	pulldowns	(The source or drain is connected to GND)
	pullups	(The source or drain is connected to VDD)
d-transistors		
	pullups	(The gate and the source are the same **and** the drain is connected to VDD **or** the gate and the drain are the same **and** the source is connected to VDD)
	super buffer	(The gate and the source **and** the gate and the drain are not the same)
	funny	(The source and the drain are the same **or** either the source or the drain are connected to GND **or** neither the source or the drain is connected to VDD)
	fixed gate	(The gate is connected to VDD or GND)
p-transistors		
	funny	(The gate and the drain **or** the gate and the source **or** the source and the drain are the same **or** the gate or the drain is connected to GND)
	pullups	(The drain or source is connect to VDD)
	fixed gate	(The gate is connect to VDD or GND)
unknown		

If the workspace is of type GATE then a report is generated showing the number of known and unknown logicgates. Any user workspace that has the gatews as a parent will have a type of GATE. The definition of 'logicgate' will be discussed thoroughly when the 'classifygates' command is discussed below. The known logicgates are partitioned into six classes. The classes supported and a brief description of each class are listed below:

Known Logic Gates	
Dynamic	(All paths to VDD and GND are clocked)
Invalid Dynamic	(One, but not all paths to VDD and GND are clocked)
Pseudo NMOS	(Only one path to VDD and path is through a p-transistor and p-transistor is gated by GND)
Invalid Pseudo-NMOS	(Pseudo NMOS with more than one path to VDD)
NMOS	(A single path to VDD and it is through e-transistor or a d-transistor)
Invalid NMOS	(More that one path to VDD and they are through a single e-transistor or d-transistor) and path is through
Static CMOS	(None of the above)
Unknown Logic Gates	(Node contains either: one or more paths to GND or one or more paths to VDD but not both)

These classes are discussed more thoroughly when the 'classifygates' command is discussed in Section 3.2 of this appendix. Each gate has boolean parameters that are set when the gates are classified. These parameters are: 'Known', 'Dynamic', 'InvalidDynamic', 'PseudoNMOS', 'InvalidPseudoNMOS', 'NMOS', 'InvalidNMOS' and 'StaticCMOS' and can all be accessed through the gate workspace (gatews) with the 'for' command.

3.1.2 For Command

for *BooleanExpression* [from '*source workspace*'] [to '*destination workspace*']

The 'for' command takes a 'BooleanExpression' and evaluates it for each element of the source workspace. If an element in the source workspace evaluates to true, then that element is added to the destination workspace. Appendix B contains a grammatical description of 'BooleanExpression'. In the following description of the 'for' commands a number of different parameters will be used. A complete list of the allowable parameters and a brief description of each is in the glossary of terms in Appendix C. If the source workspace is not specified then the default is determined by the type of the BooleanExpression. If the type of the BooleanExpression is TRANS, NODE or GATE, then the default workspaces would be transws, nodews and gatews respectively.

If the source workspace is specified, then its type is checked for consistency with the BooleanExpression. Since the gatews is just a subset of the nodews, it is allowable for a BooleanExpression of type NODE to have a source workspace that is of type GATE. It is not valid however, for a BooleanExpression of type GATE to have a source workspace that is of type NODE.

The destination workspace is always cleared before it is used. The destination workspace inherits the type from source workspace. It is an error for the destination workspace to be a system workspace, because the user cannot overwrite a system workspace. The destination workspace must be a user workspace (w0-w7). If the destination workspace is not specified then the default is workspace w0.

The type of the BooleanExpression is determined by the parameters that are used in the BooleanExpression. There are three types of parameters:

Transistor	Transistor parameters are variables that are associated with a transistor. These parameters are part of an element in a workspace of type TRANS.
Node	Node parameters are variables that are associated with a node. These parameters are part of an element in a workspace of type NODE.
Gate	Gate parameters are variables that are associated with a logicgate. These parameters are part of an element in a workspace of type GATE.

It is allowable to mix parameters of type GATE and NODE in a BooleanExpression. The destination workspace will then have a type of GATE.

In the following 'for' statement:

    for Width > 5

The number '5' is a constant and therefore has no type. MOSSTAT allows constants to be entered as float or integer. The parameter 'Width' refers to the width of a transistor that has a type of TRANS, therefore the type of the 'BooleanExpression' is TRANS. Note that the 'BooleanExpression' is just a single 'SimpleExpression' made up of two operands and a 'RelationalOperator'. Valid 'RelationalOperators' are '=' (equal), '!=' (not equal), '<' (less than), '>' (greater than), '<=' (less than or equal), '>=' (greater than or equal). Either or both of the operands may be parameters. All computations are done in float so constants can be expressed as floats or integers. A constant expression (an expression which contains no parameters) is considered to be invalid. This 'for' statement would create a list of transistors that have a width greater than 5 centimicrons. The list would be stored in the default destination workspace (w0).

The 'for' statement:

    for (GateCap + AreaCap) / 1000 > .1

will create a list of nodes that have a total capacitance greater than 0.1 picofarads. The SimpleExpression contains a 'MulOp' and an 'AddOp'. MOSSTAT observes the normal precedence rules from algebra and allows the use of '( )'s to achieve the desired result. Valid 'MulOp's are '*' (multiply), '/' (divide). Valid 'AddOp's are '+' (add), '-' (subtract). The type of this expression is NODE because all parameters are of type NODE.

The ‘for’ statement:

```
for NodeName = a
```

contains a ‘StringExpression’ of type NODE. This will create a workspace containing all nodes that have a node name of "a" or have an alias of "a". Aliases of a node are just other strings (node names) that refer to the same node. This node is the primary node and only the primary nodes are actually inserted into destination workspace. If "a" were an alias for "VDD", then VDD would be inserted into the destination workspace. Valid ‘StringRelOp’s are ‘=’ (equal) and ‘!=’ (not equal). MOSSTAT also allows for the use of a wild-card character ‘*’ as a post-fix. The string "a*" would match all nodes whose name or one of their aliases start with ‘a’. The string "*" would match all NodeNames.

The ‘for’ statement:

```
for NodeName = fdbka | NodeName = fdbkb | NodeName = fdbkc
```

is a series of expression linked together boolean-operators and has a type of NODE. Valid BooleanOperators are ‘| ’ (or) and ‘&’ (and). MOSSTAT allows the user to use a ‘shorthand’ method of specifying some ‘or’ expressions:

1. The expression "GSD = a" is shorthand for "(Gate = a | Source = a | Drain = a)"
2. The expression "SD = a" is shorthand for "(Source = a | Drain = a)"
3. The expression "NodeName = a b" is shorthand for "(NodeName = a | NodeName = b)"

Either the ‘for’ statement:

```
for Gate = fdbka | Source = fdbka | Drain = fdbka \
| Gate = fdbkb | Source = fdbkb | Drain = fdbkb \
| Gate = fdbkc | Source = fdbkc | Drain = fdbkc
    or
for GSD = fdbka fdbkb fdbkc
```

will generate list of transistors that have their gate, source or drain connected to node "fdbka", node "fdbkb" or node "fdbkc".

The ‘for’ statement:

```
for Input = TRUE
```

will create a workspace of type NODE that contains all the nodes which have been declared as inputs. A number of parameters are boolean in nature and have values of TRUE or FALSE. MOSSTAT allows the user to use ‘T’ or ‘F’ in place of ‘TRUE’ or ‘FALSE’ respectively.

The ‘for’ command and its use of expressions are quite complex and can take considerable time to become familiar with. The user needs to become thoroughly familiar with the ‘for’ command for it is the heart of MOSSTAT’s productivity.

3.1.3 Print Command

print *[***from** *workspace] [***to** *filename] [***append** *filename]*

The 'print' command will display the entire list of elements in a workspace. If the workspace is not specified, then the default workspace is user workspace w0. If neither the 'append' nor the 'to' arguments are specified, then the output from the 'print' command is sent to 'standard out'. If the 'to' option is specified then the output is written to 'filename'. If the 'append' option is specified, then the output is appended to 'filename'. All paths for the 'append' and 'to' options are relative to the current directory.

By default 'print' will display all the values of the parameters that have already been calculated. The user can specify which parameters they want displayed with the 'setpf' and 'unsetpf' commands. These commands are described in the Section 3.3 of this appendix.

3.2 Rule Check Commands

The Rule Check Commands add information to the 'database'. These rule checks are performed on the whole design and the results are stored in the 'database'. Once this information has been stored in the 'database' it can be retrieved with the Query Commands described above.

Most rule checks set a system flag when they are executed, and if an attempt is made to execute a rule check a second time, MOSSTAT will remind the user that the command has already been run and MOSSTAT will return to the prompt. This is due to the fact that the algorithms that have been used to implement some of these rule checks cannot be run twice. The 'reset' command described in section 3.3 must be executed in order to execute the rule checks a second time.

3.2.1 Classifygates Command

class[ifygates]

The 'classifygates' command will search for nodes that can be classified as 'logicgates'. A logicgate is the output node of a functional block that implements a boolean function. This functional block is a set of transistors and nodes and may be a simple inverter or a complex nand-nor function. MOSSTAT may overlook some logicgates due to the algorithm that it uses to locate logicgates. It is unlikely, however that MOSSTAT will incorrectly identify a node as a logicgate when it is not. The algorithms that are used to identify logicgates are described thoroughly in Section 5 of "MOSSTAT: An Interactive Static Rule Checker for MOS VLSI Designs.

The classification of the logicgates is broken up into several distinct steps:

1. **Identify Power Network**. The power net is the network of nodes that are in the paths from a logicgate to VDD and GND. The power net is made up of the vddnet, which is all the nodes in the paths from a logicgate to VDD and the gndnet, which is all the nodes in the paths from a logicgate to GND. The vddnet is identified by propagating vddnet out from VDD through all transistors until a node is encountered that is connected to the source or drain of an e-transistor. This node is the edge node of the vddnet and it is marked as being an invalid logicgate. The gndnet is identified by propagating gndnet out from GND through all transistors until a node is encountered that is connected to the source or drain to an p-transistor or until a node is encountered that is in the VddNet. This node is the edge node of the gndnet and it is marked as being an invalid logicgate. If a gate structure has not already been allocated for the nodes one is allocated at this time. The schematic in figure A.1 is an example of complimentary logicgate. Nodes 'A' and 'Out' would be in the vddnet, and nodes 'B' and 'Out' would be in the gndnet. All nodes which are members of the gndnet have their 'GndNet' parameters set TRUE. All nodes which are members of the vddnet have their 'VddNet' parameters set TRUE.

2. **Specify Direction in Power Net.** The 'direction' is specified at each transistor in each vddnet and gndnet whose edge node is a invalid logicgate. In the vddnet the direction goes from VDD along each path to the logicgate. In the gndnet the direction goes from GND along each path to the logicgate. A path is a unique set of transistors in series that form a path to GND or VDD. Any transistors that are in parallel are considered to be part of two distinct paths to GND or VDD. Once the direction has been specified, the transistors in each path to VDD and GND are linked together to allow for quick analysis along these paths. In Figure A.1 there are two paths to VDD: 'Out'-'A'-'VDD' and 'Out'-'A'-'VDD'. There are also two paths to GND 'Out'-'B'-'GND' and 'Out'-'GND'. The direction through each transistor is signified with an arrow. The 'direction' at each transistor can be displayed with the 'print' command.

3. **Identify Known Logic Gates.** The node workspace (nodews) is then searched for invalid logicgates. Any logicgate that is a member of both the vddnet and the gndnet is now marked as being a valid logicgate. All nodes which are not in both the vddnet and the gndnet are left marked as invalid logicgates. Both the unknown and known logicgates are added to the logicgate workspace (gatews). The node "Out" in Figure A.1 would be marked as a valid logicgate.

4. **Calculate Minimum and Maximum Resistance Paths**. Now that we have identified all the paths from VDD and GND, identified we can calculate the resistance of each path to VDD and GND. The maximum resistance to GND, minimum resistance to GND, maximum resistance to VDD, and minimum resistance to VDD are all identified and stored in the 'MaxRtoGnd', 'MinRtoGnd', 'MaxRtoVdd' and 'MinRtoVdd' parameters. These are all gate parameters and must be accessed through the gate workspace (gatews) with the

'for' command. In Figure A.1 if we assume that all the transistors have one square of resistance, the MinRtoVdd, MaxRtoVdd would both be equal to two squares. In the gndnet the MinRtoGnd will be one square and the MaxRtoGnd will be two squares.

5. **Classify Gates.** Next the valid logicgates are classified into one of following groups:

    Dynamic. If all of the paths from VDD and GND to the logicgate are 'clocked', paths then the logicgate is classified as dynamic. A clocked path is a path where at least one of the transistors in the path is gated by a clocked node. A clocked node is a node that has been declared as a clock or is classified as a valid dynamic logicgate. In Figure A.2, if node 'phi1' is a clock the node 'Out' is a dynamic logicgate.

    Invalid Dynamic. If some but not all of the paths from VDD and GND to the logicgate are clocked paths then the logicgate is classified as an invalid dynamic gate.

    Pseudo NMOS. If the logicgate has a single path to VDD and that path is through a single p-transistor and that p-transistor is gated by GND, then the logicgate is classified as a pseudo NMOS gate.

    Invalid Pseudo NMOS. If the logicgate has more than one path to VDD and at least one of those paths is a single p-transistor that is gated by GND, then the logic is classified as an invalid pseudo NMOS gate.

    NMOS. If a logicgate has a single path to VDD and it is through a single e-transistor or single d-transistor then it is classified as an NMOS Gate.

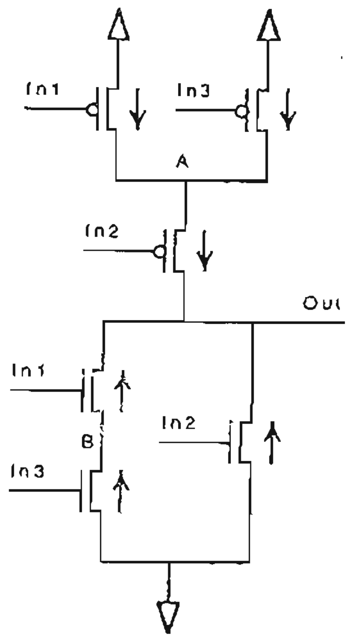
    Invalid NMOS. If a logicgate has more that one path to VDD and it is through a single e-transistor or a single d-transistor then it is classified as an Invalid NMOS Gate.

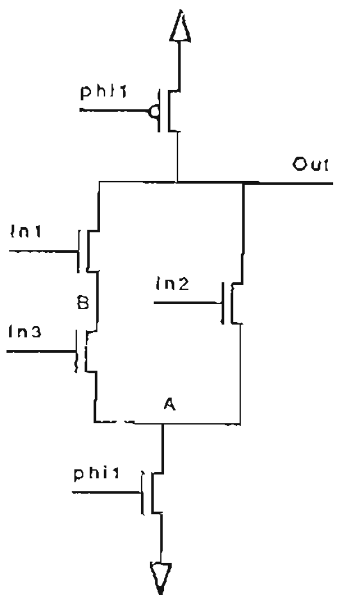
    Static CMOS. If a logicgate does not fall into any of the above classifications then it is classified as an Static CMOS gate.

If the logicgate is classified as dynamic then, MOSSTAT will calculate the 'net capacitance'. The net capacitance is the total capacitance of all the nodes in the gndnet and the vddnet with the exception of VDD, GND and the logicgate itself. The result is stored in the 'NetCap' parameter. This gate parameter can be accessed through the gate workspace (gatews). The net capacitance is needed to enforce the third rule of the Dynamic Clocking Rules (see appendix C).

The logicgate in Figure A.1 will be classified as static CMOS. The logicgate in Figure A.2 will be classified as dynamic CMOS. In Figure A.2 the net capacitance in the dynamic gate will be the sum of the area capacitance of node 'A' and node 'B'. In practice the net capacitance will often be zero because the nodes in the power network are not connected to any transistor gates and will generally have negligible area capacitance. Since we are looking for a ratio of the capacitance of the logicgate to the net capacitance, a net capacitance of zero would give a infinite ratio. MOSSTAT therefore assumes that the minimum net capacitance is .1 femtofarads. This means that a ratio of five to one is achieved if the total capacitance of the logicgate is .5 femtofarads. Anything less than .5 femtofarads would be violation of the Dynamic Clocking Rules.

The 'classifygates' command can only be run once. In order to change the input, output or clock declarations either MOSSTAT must be restarted or the 'reset' command must be run. This command is discussed in Section 3.3 of this appendix.





Figures A.1 and A.2

3.2.2 Effects Command

eff[ects]

The 'effects' command marks all the nodes which are effected by an input. It then marks all the nodes that can effect an output. No report will be generated, but results are stored in the database. The boolean parameters that hold the results of the 'effects' command are 'EInput' and 'EOutput'. These are node parameters and are accessible through any workspace that is of type GATE or NODE. The reasoning behind this rule check is that any node that cannot effect the outputs or is not effected by the inputs cannot effect the functionality of a design and are therefore unnecessary logic. This unnecessary logic may occur due to the way a circuit is laid out or it may be due to improper specification of the circuit.

The 'effects' command can only be run once. The 'effects' command cannot be run until the inputs, outputs and the clocks have been declared. In order to change the input, output or clock declarations and re-execute the 'effects' command, either MOSSTAT must be restarted or the 'reset' command must be run. The 'reset' command is discussed in Section 3.3 in this appendix Miscellaneous Commands.

3.2.3 Propagate Command

prop[agate]

The 'propagate' command marks all the nodes that can be set either high or low. All inputs and VDD initially have their high bits set. All inputs and GND initially have their low bits set. The high bits and low bits that have been set are propagated from the source to the drain of a transistor(or visa versa) when any of the following conditions is met:

1. The transistor is a p-transistor and the node connected to its gate has its low bit set.
2. The transistor is a e-transistor and the node connected to its gate has its high bit set.
3. The transistor is a d-transistor.

The low bit for VDD and the high bit for GND cannot be set. In this fashion the bits are propagated until no more changes occur. Any nodes that do not have the high bit set after this evaluation cannot be set high. Any nodes that do not have the low bits set after this evaluation cannot be set low. The node parameters associated with these bits are 'High' and 'Low'. These parameters are accessible with the 'for' command.

The 'propagate' command cannot be run until the inputs, outputs and clocks have been declared. In order to change the input, output or clock declarations and re-execute the 'propagate' command, either MOSSTAT must be restarted or the 'reset' command must be run. The 'reset' command is discussed in Section 3.3 of this appendix.

3.2.4 Thresholds Command

thresh[olds]

The concept of a threshold drop comes from the physical nature of an e-transistor. Even when an e-transistor is fully turned on (its gate is high), it can only pull its source to within 1 threshold drop of its gate voltage. On the other hand it is capable of pulling its drain all the way to zero when it is turned on. The range of voltage at the drain is 0-4 volts. When a d-transistor is fully turned on (its gate is greater than or equal to negative 3 Volts), it is capable if pulling its source high with no threshold drops.

MOSSTAT introduces the concept of a threshold rise that comes from the physical nature a p-transistor. When a p-transistor is fully turned on (its gate is low), it can only pull its source down to within 1 threshold rise of its gate voltage. A p-transistor is, however, capable of pulling its drain all the way high when it is enabled. The voltage at the source is 1-5 volts. An e-transistor is effective at passing a low signal and a p-transistor is effective at passing a high signal. It is for that reason that CMOS uses transmission gates consisting of both an e-transistor and a p-transistor.

When the 'thresholds' command is executed, MOSSTAT will calculate the threshold drops and rises at each node. The behavior of threshold drops can be characterized with the following formulas:

d-transistors
Source Threshold Drops = max(GateThreshDrops-3, DrainThreshDrops)
e-transistors
Drain Threshold Drops = max(GateThreshDrops+1, SourceThreshDrops)
p-transistors
Drain Threshold Drops = SourceThreshDrops

Initially all nodes except VDD and any nodes that have been declared to be inputs are assumed to have ten threshold drops. All inputs and VDD are assumed to have zero threshold drops. These formulas are repeatedly applied until the threshold drops at each node can be reduced no further and no more changes occur. Ten was chosen because it is more than the largest possible number of threshold drops on a node.

MOSSTAT will also calculate the threshold rises at each node. Threshold rises do not need to be calculated across d-transistors due to the fact that MOSSTAT assumes that d-transistors have their drains connected directly to VDD. The behavior of threshold rises can be characterized with the following formulas:

e-transistors
DrainThresholdRises = SourceThreshRises
p-transistors
DrainThresholdRises = max(GateThreshRises+1, SourceThreshRises)

Initially all nodes except GND and any nodes that have been declared to be an input are assumed to have ten threshold rises. All inputs and GND are assumed to have zero threshold rises. These formulas are repeatedly applied until the threshold drops at each node can be reduced no further and no more changes occur.

No report is generated but the results of the 'thresholds' command are stored in the database in the integer parameters 'ThreshDrop' and 'ThreshRise'. These parameters are accessible to the user with the 'print' or the 'for' command. The 'thresholds' command can only be run once. The 'thresholds' command cannot be run until the inputs, outputs and clocks have been declared. If there is a need to change input, output or clock declarations and re-calculate the threshold drops and rises either MOSSTAT must be restarted or the 'reset' command must be run.

Typically in CMOS designs the designer does not want any threshold rises or drops. Threshold drops are allowed on nodes which are in paths from a logicgate to GND and that path is through a series of e-transistors. As an example the logicgate in Figure A.2 nodes 'A' and 'B' which each have one threshold drop but this is valid because these nodes are not connected to the gate of a transistor. The same is true for threshold rises through a network of p-transistors to VDD. In the logicgate is Figure A.1 node 'A' is valid because 'A' is not connected to the gate of a transistor. In NMOS designs 1 threshold drop is allowable on nodes that are connected to gates of transistors.

3.3 Miscellaneous Commands

Following is a list of all the remaining commands for MOSSTAT. These commands are the ones that can not be classified as either Rule Check Commands or Query Commands. They are listed in alphabetical order.

alias *NodeNameList*

List all the node names which are aliases for each node in the 'NodeNameList'. A 'NodeNameList' is a list of node names separated by spaces. The first node name in the output list is the primary node and all the other node names are aliases to it.

clock[s] *[NodeNameList]*

Declare all the nodes in the 'NodeNameList' to be clocks. Any node which is a clock is also declared to be an input. A 'NodeNameList' is a list node names separated by spaces. The 'clocks' command can be run more than once to add additional clock declarations. All clock declarations are discarded when the 'reset' command is executed. If the 'NodeNameList' is omitted, MOSSTAT will print a list of the nodes that have been declared as clocks.

display *NodeName*

Display the node structure. The 'display' command is primarily a debugging aid. The 'display' command ignores any aliases and will display the actual structure associated with the NodeName. If NodeName is an alias, then it should be empty with the exception of the node name field and the pointer to the primary node. The output is cryptic and the user needs to look at the source for the procedure Display() in commands4.c in order to make sense out of the output. The user will need to look for the declaration of NODESTRUCT in structures.h. This command is only intended to give assistance to anyone needing to make internal changes to

MOSSTAT.

help *[CommandNameList]*
help parameter

The 'help' facility will provide the user a minimum amount of information about a command. A grammatical description of 'CommandNameList' is contained in Appendix B. If 'CommandNameList' is not specified 'help' will provide the user with a list of all the possible commands. If 'CommandNameList' is provided, MOSSTAT will print out the syntax of the commands listed. If the keyword 'parameter' is specified then a list of all the possible parameters is generated.

input[s] *[NodeNameList]*

Declare all the nodes in the 'NodeNameList' to be inputs. The 'inputs' command can be run more than once to add additional input declarations. All input declarations are discarded when the 'reset' command is executed. If the 'NodeNameList' is omitted, MOSSTAT will print a list of the nodes that have been declared as inputs.

output[s] *[NodeNameList]*

Declare all the nodes in the NodeNameList to be outputs. The 'outputs' command can be run more than once to add additional output declarations. All output declarations are discarded when the 'reset' command is executed. If the 'NodeNameList' is omitted, MOSSTAT will print a list of the nodes that have been declared as outputs.

path[s] *NodeNameList*
path[s] *[WorkSpaceList]*

List the nodes in all known paths from each logicgate to VDD and GND and specify the resistance of these paths in squares. Since analysis in done statically, the resistance calculation assumes that only one path is enabled at a time. Work-space-list is a list of workspaces separated by spaces. The workspaces in the WorkSpaceList must be of type GATE. If no workspace is specified, then the default workspace is w0. NodeNameList is a list of nodes separated by spaces that are to be evaluated. If any node in the NodeNameList is not a valid logicgate, it is ignored.

quit

Exit from MOSSTAT and return to the shell. All information in the data base is destroyed.

reset

Reset MOSSTAT. The 'reset' command returns MOSSTAT to its start-up state after it has read in the simfile and the alias file and created the internal data base. Inputs, outputs and clocks must be re-declared. The Rule Check Commands: 'classifygates', 'effects', 'propagate' and 'thresholds' can now be run again. Any new aliases that were created with the 'setalias' command are not destroyed and do not need to be re-declared.

setalias *primary-node-name alias-node-name*

Create a new alias for a node. This is not the same as a connect command which would allow you to connect two or more existing nodes. You also cannot create any new nodes. You may, however, create a new alias to an existing primary node or alias an existing primary node and all its aliases to a new primary node. You must refer to any existing nodes by their primary node name. If you use a alias node name MOSSTAT will issue a warning message. For example assuming that "fdbka" is an existing primary node name you cannot:

setalias gnd fdbka

Because "gnd" is an alias node name for the primary node "GND" and you also cannot:

setalias GND fdbka

Because "fdbka" and "GND" are both primary node names and you cannot connect two existing nodes.

setpf *[PrintFlagList]*
unsetpf *[PrintFlagList]*

Specify a list of parameters that will be displayed by the 'print' command. A PrintFlagList is a list of printflags separated by spaces. A printflag is an integer from one to eleven that specifies the print status of one or more parameters. The value of the printflag determines whether or not the 'print' command will display the value of the particular parameter. By default the 'print' command will display all the parameters that have been calculated. When a 'setpf' or and 'unsetpf' is executed the status of the print-flags remain unchanged until another 'setpf' or 'unsetpf' command is executed. A call to 'setpf' will set all the printflags in the PrintFlagList TRUE. A '*' in the PrintFlagList of a 'setpf' command will set all the printflags TRUE. A call to 'setpf' or 'unsetpf' with no printflags will print the current state of the printflags. A call to 'unsetpf' will set all the printflags in the PrintFlagList FALSE. A '*' in the PrintFlagList of a 'unsetpf' command will set all the printflags FALSE.

The following is a list of the parameters and the print-flags that are associated with each parameter:

1	Input Output Clock	(NODE)
2	EInput EOutput High Low	(NODE)
3	GndNet VddNet	(NODE)
4	ThreshDrop ThreshRise	(NODE)
5	GateCap AreaCap	(NODE)
6	Logic Gate Classification	(GATE)
7	MaxRtoGnd MinRtoGnd MaxRtoVdd MinRtoVdd	(GATE)
8	NetCap .	(GATE)
9	Direction	(TRANS)
10	Length Width	(TRANS)
11	Area	(TRANS)
12	PrintCode	

Printflag twelve is a debugging feature and by default it is turned off. When printflag twelve is set, the RPN Code Stack will be printed every time the 'for' command is executed.

source *file-name*

A valid file name can contain letters(a-zA-Z), digits(0-9), underscore(_), dot(.), or slash (/). Any printable character is valid if the file-name is inside double quotes ("). The file-name is loaded and the contents interpreted by the Mosstat Interactive Environment. The file is expected to contain valid interactive commands. All text between a leading '(' and a trailing ')' is treated as a comment. The actions of the interpreter are exactly the same as they would be if the commands were typed into the terminal interactively. The file can contain another 'source' command but beware of reference loops. If the file contains a 'quit' command then MOSSTAT will exit immediately. If the end-of-file is reached then control of MOSSTAT will return to the terminal and MOSSTAT will respond with a '.'.

sys[tem] *CommandStringList*

The 'system' command passes the CommandStringList to sh(1) as input, as if the string had been typed as a command at the terminal in an interactive shell. The 'system' is a simple minded implementation and is not meant to give the user full access to the UNIX shell. The grammatical description of 'CommandNameList' is Appendix B. Simple things like grep, cat, more, pipes, redirection are allowed. A valid command can contain letters(a-zA-Z), digits(0-9), underscore(_), dots(.), or slash(/).

4. Mosstat Methodology and Examples

This section has been included to give the user an opportunity to see how MOSSTAT could be used in practice. This section outlines the approach that MOSSTAT allows the user to take when trying to detect and isolate errors in a circuit design. A number of examples will be given that will clarify this approach.

4.1 Mosstat Methodology

Dynamic analysis tools are designed to help the user detect logic errors in a circuit design. Some of these tools can be used to get additional timing information from the design. These tools can also be use to detect specification errors in the circuit design. The goal of static analysis tools such as MOSSTAT is to isolate specification errors in the design. This duplication of functionality does not lessen the need for static analysis tools. By its very nature static analysis is more simplistic than dynamic analysis and, therefore, static analysis tools such as MOSSTAT have the potential to detect and isolate errors at a much lower cost.

Errors such as design rules violations can be detected by careful inspection of a plot of the design. This tedious inspection has been largely replaced by design rule checkers because effective algorithms have been formulated for design rule error detection. Careful inspection of the netlist by the designer can be used to detect electrical rule violations. Like hand checking for design rules in a physical layout, this is time consuming and error prone. The difficulty is that electrical rules have a tendency to be more abstract than physical design rules. The abstract nature of these design rules makes it more difficult to provide effective algorithms that can be used to detect electrical rule violations. MOSSTAT provides the user with a number of rule checks that can aid in the detection and isolation of errors. In addition, MOSSTAT will provide the designer with an environment that makes the job of analyzing the circuit quicker and less error prone. MOSSTAT does this by providing the following three methods:

1. Provide a method for characterizing nodes, transistors or logicgates with respect to a specific set of rules. These characterizations allow the user to break up the design into smaller more workable groups. These smaller groups may depending on context contain nodes, transistors or logicgates that need to be further analyzed.

2. Provide a method for isolating portions of the circuit into workspaces that can be easily analyzed by the designer. These workspaces are groups of nodes, transistors or logicgates that are created by queries of other workspaces for characteristics that are specified by the designer.

3. Provide a method for analysis of these transistors, nodes or logicgates which is quick, allows feedback, and minimizes the chance for error.

MOSSTAT has a set of 'rule checks' that makes it easy to detect and isolate a number of different types of errors. The results of the 'rule checks' characterize a particular node, transistor or logicgate element. The characteristics are stored as parameters associated with each element in the database. A workspace can be created that contains the elements whose characteristics are specified by the user. The characteristics of a workspace are specified with a BooleanExpression in a 'for' command. The 'for' statement is the heart of a powerful expression-based interactive query language that allows the user to enforce many rules that are not specifically checked by the 'rule check' commands. Any parameters can be used in the boolean-expression of a 'for' command to create a workspace that contains elements of the desired characteristics.

The interactive nature of MOSSTAT allows the user to selectively execute any number of different rule checks. These rule checks can be executed in any order once all the inputs and outputs have all been declared.

4.2 Running MOSSTAT: A Typical Scenario

MOSSTAT expects to read a netlist circuit description from a valid simfile. This simfile may have been created by extraction from either schematics or physical layout. The approach that one takes to verify a circuit description is the same regardless of the source of the simfile.

Assuming that the netlist description is a simfile that is named foo.sim, MOSSTAT can be invoked by typing the following into a UNIX shell:

mosstat foo.sim

or

mosstat foo

Either command will cause MOSSTAT to begin execution and to create the internal database from the netlist description in foo.sim. MOSSTAT then reads in the file foo.al for alias descriptions and during this operation MOSSTAT will respond with the following:

```
MOSSTAT Version 1.0
Opening foo.sim for reading
Opening foo.al for reading
```

Next MOSSTAT will check for the existence of VDD and GND. If either is missing, MOSSTAT will issue an error message and exit. Otherwise, MOSSTAT will respond with a '.'. The '.' is the prompt and it means that MOSSTAT is ready to receive commands from the terminal.

A good place to start is by asking for a count of the 'nodews' (the node workspace): This is done with the 'count' command:

count nodews

MOSSTAT will display a count of the total number of nodes and of the number of nodes that have aliases. The user can check these numbers for plausibility. Next we could get a count of the 'transws' (the transistor workspace) with the 'count' command:

count transws

MOSSTAT will display counts of the numbers of transistors of each type. Transistors of the same type are further classified as funny, pullup, etc. These numbers should also be checked for plausibility. If any transistors are classified as funny or fixed gate they should be further investigated. Section 3.1.1 provides a complete description of the different classifications of each transistor type. Isolating groups of transistors is best done with a query using the 'for' command. If, for example, MOSSTAT classifies several transistors as funny p-transistor, we can create a workspace containing all those transistors with the following 'for' command:

```
for type = p-transistor & ( gate = drain | gate = source\
| source = drain | gate = VDD | gate = GND )
```

A default workspace (w0) is then created which contains all the p-type enhancement transistors that are classified as funny. Similar techniques can be used to further investigate any transistor classifications. This workspace can be printed with either of the following commands:

print
  or
print w0

When the 'print' command is executed, MOSSTAT will check the values of the printflags to determine which parameters are to be displayed. These flags are set with the 'setpf' and the 'unsetpf' commands. By default these flags are all turned off and therefore the 'print' command will display the transistor type and a list of the nodes that are connected to the Gate, Source and Drain of each transistor.

The next step is to specify the inputs, outputs and clocks. This can be done by typing them into the terminal, but an easier and less error prone method is to put them into a file. This file can be executed at MOSSTAT initialization by specifying the -c command line option to MOSSTAT or with the 'source' command from within MOSSTAT. To declare the inputs, outputs and clocks from the terminal:

inputs in1 in2 in3; outputs out1 out2 out3; clocks phi1 phi2

If the commands are already in the file 'declareIO':

source declareIO

Once the inputs and the outputs have been declared we can run the 'effects' and the 'propagate' commands. When the 'effects' and 'propagate' commands are executed MOSSTAT sets the 'High', 'Low', 'EInput' and 'EOutput' node parameters. The validity of these parameters can then be checked with the use of the of the 'for' command.

For example, the following 'for' command:

for High = FALSE / Low = FALSE

would create a workspace that would contain all the nodes that cannot be set high or low. A similar command could be used to create a workspace that contained the nodes which could not effect the inputs or could not effect an output:

for EInput = F / EOutput = F

If it becomes apparent that some inputs or outputs were not declared it is possible to add them at this time. This is done by first executing the 'reset' command. After the 'reset' command the inputs, outputs and clocks must be declared again along with the new inputs and outputs; then the 'effects' and 'propagate' commands can be executed again.

The next step is to run the 'thresholds' command. The number of threshold drops that are allowed depends on the technology being used. To create a workspace that contains all the nodes with more than one threshold drops the following 'for' command could be used:

for ThreshDrop > 1

A similar 'for' command could be used to find the nodes that have more than zero threshold rises.

The next command to be run is the 'classifygates' command. The 'classifygates' command will create a new system workspace, the gatews. The 'count' command can then be executed on the gatews:

count gatews

The user can further evaluate the classifications of the logicgates with the 'for' command. The following 'for' command:

for InvalidDynamic = TRUE

would create a workspace containing all the logicgates that were classified as Invalid Dynamic CMOS.

When the 'classifygates' command was executed the minimum and maximum resistance to GND and VDD was calculated for each logicgate. These parameters can be used to estimate the rise and fall times for each logicgate. The following 'for' command:

*for ((MaxRtoGnd * 10 * (AreaCap + GateCap)) / 1000) > 9*

will create a workspace that contains all the logicgates that have a fall time that is greater than 9 nanoseconds (ns). 'MaxRtoGnd' is the Maximum Resistance to GND in squares. '10' is the resistance of an e-transistor in K-ohms per square. 'AreaCap' and 'GateCap' is the area and gate capacitance of the logicgate the have units of picofarads (pf). The 1000 is a constant give a result in nanoseconds.

The following 'for' command:

for MaxRtoVdd / MinRtoGnd < 4

will create a workspace that contains all the logicgates that have paths with ratios less than 4.

Possible charge-sharing problems can be detected with the following 'for' expression:

*for GateCap + AreaCap < 5 * NetCap*

This expression will create a workspace that contains all the logicgates that have storage nodes with less than five times the combined capacitance of all the nodes in the GndNet and the VddNet. All the logicgates which are in violation of the Dynamic Clocking Rules are contained in this workspace. The 'GateCap', 'AreaCap' and 'NetCap' parameters all have units of picofarads (pf).

Appendix B: The Grammar for a Boolean Expression

Below is the grammar that describes a BooleanExpression for the 'for' command. Terminal symbols are either in bold letters or are non-alphabetic characters enclosed in quotes (''). Although the syntax used to describe this grammar is similar to that used in the YACC specification, it is not meant to be an language description. The complete definition of the grammer for the interactive environment is in the source file 'grammar' which contains the YACC specification for the parser.

```
BooleanExpression :
        Expression
        | BooleanExpression BooleanOperator BooleanExpression
        ;
Expression :
        StringParameter StringRelOp NodeNameList
        | SimpleExpression RelOp SimpleExpression
        ;
SimpleExpression :
        Integer
        | Float
        | TRUE | T
        | FALSE | F
        | Parameter
        | SimpleExpression MulOp SimpleExpression
        | SimpleExpression AddOp SimpleExpression
        ;
StringParameter :
        Type
        | NodeName
        | Gate
        | Source
        | Drain
        | SD
        | GSD
        ;
NodeNameList :
        NodeName
        | NodeName NodeNameList
        ;
```

```
NodeName :
        NodeString
        | NodeString'*'
        | "NodeString"
        | "NodeString'*'"
        | '*'
        ;
Parameter :
        NodeParameter
        | GateParameter
        | TransistorParameter
        ;
NodeParameter :
        Input
        | Output
        | Clock
        | EInput
        | EOutput
        | High
        | Low
        | GndNet
        | VddNet
        | ThreshDrop
        | ThreshRise
        | GateCap
        | AreaCap
        | Known
        ;
GateParameter :
        Pseudo
        | InvalidPseudo
        | Dynamic
        | InvalidDynamic
        | NMOS
        | InvalidNMOS
        | StaticCMOS
        | MaxRtoGnd
        | MinRtoGnd
        | MaxRtoVdd
        | MinRtoVdd
        | NetCap
        ;
```

```
TransistorParameter :
      Length
      | Width
      | Area
      | Direction
      ;
StringRelOp :      ( String Relational Operators )
      '='
      | '!='
      ;
RelOp :            ( Relational Operators )
      '='
      | '>'
      | '>'
      | '>='
      | '<='
      | '!='
      ;
BooleanOperator :
      '&'
      | '|'
      ;
AddOp :
      '+'
      | '-'
      ;
MulOp:
      '*'
      | '/'
      ;
```

The precedence of the Operators is as follows:

Multiplication Operators	'*' '/'
Addition Operators	'+' '-'
Relational Operators	'=' '<' '>' '>=' '<=' '!='
Boolean Operators	'&' '|'

Within the same precedence level, items are evaluated left to right. The normal precedence of operators is observed. Brackets ('{' & '") can be used wherever needed to get the desired order of execution. The precedence of the operators is the same as the 'C' programming language. For historical reasons a node name may be any string of printable characters (a 'NodeString'). If a node name matches a reserved word in MOSSTAT, then the node name must be put inside double quotes (a QuotedString). A '*' appended to a prefix string will match any node name whose prefix matches the prefix string. A single '*' will match all node names. A 'NodeString' can contain any printable character. An 'Integer' can be any digit, A 'Float' can contain a decimal point but cannot contain an exponent. The scanner will only recognize a 'NodeString' if it is parsing an expression that contains a 'StringParameter' followed by a 'StringRelOp'. This frees the user from including white space as token separators in all expressions except when specifying a 'NodeNameList'. A 'Integer' is 1-N digits optionally preceded by a sign. A 'Float' can be 1-N digits optionally preceded by a sign and may contain a decimal point.

Below is the grammatical description for WorkSpaceList and CommandNameList.

```
WorkSpaceList :
        WorkSpace
        | WorkSpace WorkSpaceList
        ;
WorkSpace :
        transws
        | nodews
        | gatews
        | w0 | w1 | w2 | w3n | w4 | w5 | w6 | w7
        ;
CommandNameList :
        CommandName
        | CommandName CommandNameList
        ;
CommandName :
        count                           /* Query Commands               */
        | for
        | print
        | classifygates | class         /* Rule Check Commands          */
        | effects | eff
        | propagate | prop
        | thresholds | thresh
        | alias                         /* Miscellaneous Commands       */
        | clocks
        | help
        | inputs
        | outputs
        | paths
        | quit
        | setpf
        | unsetpf
        | source
        | system | sys
        | unsetpf
        ;
```

Appendix C: Glossary of Terms

BooleanExpression

A BooleanExpression is an expression that may contain constants, parameters, string constants, or string parameters. Associated with a BooleanExpression is a TYPE and that type is determined by the types of the parameters in the BooleanExpression. A BooleanExpression that is of type NODE can only contain parameters of type NODE. A BooleanExpression that is of type GATE can contain parameters of type NODE or GATE. A BooleanExpression that is of type TRANS can only contain parameters of type TRANS. Appendix B describes the grammar of a BooleanExpression in more detail.

database

The complete description of the circuit is contained in the database. All information created by interactive rule checks is stored in this database. The organization of the data elements in the data allows extremely fast access to the information in the database. The contents of the database are accessible with the 'print' command.

Dynamic Clocking Rules

Clocking for Dynamic MOS Systems [Bra86] is governed by three simple rules, which, when followed, guarantee correct, race-free operation of such systems, the are:

1. Inputs to combinational logic should never change while the enabling clock is high.
2. All combinational logic should provide adequate charge storage at each input.
3. All combinational logic should provide active, restoring drive at each output.

These rules are restrictive, in that many circuits violating these rules will still operate reliably. If followed carefully, however they will ensure that CMOS circuits using two-phase, non-overlapping clocks will never suffer from inadvertent charge-sharing.

gatews

Gate workspace: See entry on workspace below.

GndNet and VddNet

In order for MOSSTAT to find a logicgate it must isolate all of the paths from that logicgate to VDD. Any node that is a part of one of these paths is a member of the VddNet. MOSSTAT must also isolate all the the paths to GND, and any node in one of those paths is in the GndNet. A valid logicgate is a member of both a GndNet and VddNet.

logicgate

A logicgate is the output node of a functional block that implements a boolean function. This functional block is a set of transistors and nodes and may be a simple inverter or a complex nand-nor function. MOSSTAT may miss some logicgates due to the limitations of the algorithm that it used to locate them. It is unlikely, however, that MOSSTAT will incorrectly identify a node as a logicgate when it is not. A detailed description of the methodology of identifying logicgates is in Section 5.3.

NetCap

The NetCap is a parameter associated with each logicgate. It is the total capacitance of all the nodes in the GndNet and the VddNet with the exception of VDD, GND, and the logicgate itself (the intersection of the GndNet and the VddNet). This parameter is needed to check for rule 3 of the Dynamic Clocking Rules.

nodews

Node workspace: See entry on workspace below.

parameter

A parameter is a 'variable' in a BooleanExpression that refers to a field in an entry in the workspace. Transistor, node and gate parameters are of type TRANS, NODE and GATE respectively. Parameters of type NODE and GATE can be used together in an expression if the source workspace is of type GATE. All the parameters are displayed as either a boolean, integer or float. A StringParameter can either of type NODE or TRANS and will be displayed as a string. The type associated with a parameters is used to enforced the consistency of access to a workspace. The value of a parameter for a particular node or transistor can be displayed with the 'print' command. The type of a BooleanExpression is determined by the type of the parameters that are in the BooleanExpression. A BooleanExpression can only be of type NODE, TRANS or GATE.

The following is a list of the node parameters with a brief description of each and its data type. All boolean are stored as flags that are set TRUE or FALSE. The 'Direction' in a transistor is a tri-state variable and is either SourceToDrain, DrainToSource or unset. The user may type 'TRUE' or 'T' for true and 'FALSE' or 'F' for false.

NodeName	(NodeName, string)
Input	(Node is an input, boolean)
Output	(Node is an output, boolean)
Clock	(Node is a clock, boolean)
EInput	(Node is effected by an input, boolean)
EOutput	(Node is effected by an output, boolean)
GndNet	(Node is in GndNet, boolean)
VddNet	(Node is in VddNet, boolean)
ThreshDrop	(Number of threshold drops at node, int)
ThreshRise	(Number of threshold rises at node, int)
GateCap	(Capacitance on node due to Gates, femto-fahrads, float)
AreaCap	(Capacitance on node due to Area, femto-fahrads, float)

The following is a list of the gate parameters with a brief description of each.

Known	(Node is a logicgate of known type, boolean)
Pseudo	(Node is a pseudo-NMOS logicgate, boolean)
InvalidPseudo	(Node is an invalid pseudo-NMOS logicgate, boolean)
Dynamic	(Node is dynamic logicgate, boolean)
InvalidDynamic	(Node is an invalid dynamic logicgate, boolean)
NMOS	(Node is an NMOS logicgate, boolean)
StaticCMOS	(Node is a static CMOS logicgate, boolean)
MaxRtoGnd	(Maximum Resistance to Gnd, squares, integer)
MinRtoGnd	(Minimum Resistance to Gnd, squares, integer)
MaxRtoVdd	(Maximum Resistance to Vdd, squares, integer)
MinRtoVdd	(Minimum Resistance to Vdd, squares, integer)
NetCap	(Net Capacitance of logicgate, femto-fahrads, float)

The following is a list of the Transistor parameters with a brief description of each.

Type	(Transistor Type, string)
Gate	(Name of Node connected to Gate of Transistor, string)
Source	(Name of Node connected to Source of Transistor, string)
Drain	(Name of Node connected to Drain of Transistor, string)
Direction	(Data direction through VddNet and GndNet, tri-state)
Length	(Transistor Length, centimicrons, float)
Width	(Transistor Width, centimicrons, float)
Area	(Transistor Area, square-centimicrons, float)
GSD	(Name of Node connected to Gate, Source or Drain of Transistor, s
SD	(Name of Node connected to Source or Drain of Transistor, string)

simfile

A simfile is a netlist (transistor list) description of a MOS VLSI circuit. It contains information about transistor sizes and line capacitance. Its does not model line resistance. By convention the file name ends with a '.sim'. There are several different formats of a simfile all of which are acceptable by MOSSTAT. For the definitive description of the format of the simfile see the UNIX cadman page on simfile.

transws

Transistor workspace: See entry on workspace below.

VddNet

See entry on GndNet above.

workspace

A workspace is a 'list' of entries into the database. Each of these elements is actually just a structure in 'C'. There are two different kinds of workspaces, a User workspace and System workspace.

A System workspace is created once and can never be over written. A User workspace is a temporary list that can be created or re-used by the user with the 'for' command.

There are three system workspaces: 'gatews', 'transws' and 'nodews'. 'transws' is a list of all the transistors in the circuit and 'nodews' is a list of all the nodes in the circuit including aliases. Both 'transws' and 'nodews' are created when MOSSTAT creates the database from the net description. 'gatews' is a list of all the nodes in the circuit that are classified as logicgates and is created when the 'classifygates' command is executed by the user.

Also associated with each workspace is a type. A workspace can be a list of elements associated with nodes in the circuit, a list of elements associated with logicgates, or a list of elements associated with transistors in the circuit 'nodews' is of type NODE, 'transws' is of type TRANS, and 'gatews' is of type GATE. The type is MOSSTAT's way of keeping track if a workspace is pointing to a list of node elements, transistor elements or gate elements. A BooleanExpression can only be evaluated on a workspace whose type is the same as the type of the BooleanExpression. Any user workspace that is created inherits the type of the parent workspace.

MOSSTAT can allocate up to 8 User workspaces (w0 - w7). These workspaces can be quite large (4 Bytes per transistor), consequently the workspaces are created dynamically as they are needed.

Appendix D Mosstat Source Files

This appendix briefly describes the files contained in the MOSSTAT source directory. A convention for naming source files that should assist anyone needing to access the sources of MOSSTAT. A Table of Contents has been included that is an alphabetical list of all the procedures in MOSSTAT and the files in which they are contained. The sources for MOSSTAT are available in [Joh86].

classgates1.c classgates2.c

The procedures need to implement the 'classifygates' command.

commands1.c commands2.c commands3.c commands4.c

The procedures that implement the interactive commands. Many procedures (like ClassifyGates) will call other procedures.

dbase.c

The low level routines that interact directly with the database.

define.h

The symbolic constants for MOSSTAT. Several of the symbolic constants are machine dependent.

evalnode.c

The procedure EvalNodeCode(). This procedure evaluates an expression on a workspace of type NODE or GATE.

evaltrans.c

The procedure EvalTransCode(). This procedure evaluates an expression on a workspace of type TRANS.

global.c

The global variable and structure definitions.

grammar

The YACC source for the parser for the interactive environment.

interact1.c interact2.c

The procedures that implement the general utility functions for the interactive environment.

main.c

The main() procedure.

parse.c

The parser for the simfile and the aliasfile.

scanner.def

The LEX source for the scanner for the interactive environment.

string.c

Several procedures that implement string manipulation functions.

structures.h

All the structure declarations.

utils1.c utils2.c

Several procedures that implement general utility functions for MOSSTAT such as the Hash function, interrupt handlers, memory allocation.

y.tab.h

This file contains all the symbolic constants defined by YACC.

Table Of Contents

Procedure Name	File Name
AddNode()	dbase.c
AddTrans()	dbase.c
Alias()	commands1.c
AliasNodes()	dbase.c
Allocate()	utils1.c
AllocateAliasList()	interact2.c
AsciiToInt()	utils2.c
CalcDynamicCap()	classgates2.c
CalcResisRatios()	classgates2.c
CalcThreshDrop()	commands1.c
CalcThreshRise()	commands1.c
ClassifyGates()	commands2.c
ClassifyLogicGates()	classgates1.c
ClearCode()	interact2.c
ClearWorkSpace()	interact2.c
Clocks()	commands3.c

CompressNodeWS()	dbase.c
CompStr()	string.c
CountGates()	interact2.c
CountTrans()	interact2.c
CountWorkSpace()	commands1.c
Display()	commands4.c
Effect()	commands2.c
EffectOutput()	interact2.c
Error()	utils1.c
EvalNodeCode()	evalnode.c
EvalTransCode()	evaltrans.c
Exit()	utils1.c
FindDynamicGates()	classgates2.c
FindGates()	classgates1.c
FindGndNet()	classgates1.c
FindNmosGates()	classgates2.c
FindNodePtr()	dbase.c
FindPathsToGnd()	classgates1.c
FindPathsToVdd()	classgates2.c
FindPseudoGates()	classgates2.c
FindVddNet()	classgates1.c
For()	commands2.c
GetNode()	dbase.c
GetNodePtr()	interact2.c
GetRecord()	utils1.c
Hash()	utils1.c
Help()	commands4.c
Init_1	utils2.c
Init_2	utils2.c
input()	scanner.c
InputEffect()	interact1.c
Inputs()	commands1.c
Interrupt()	utils1.c
IsConnected()	classgates1.c
List()	commands1.c
main()	main.c
NewBuf()	utils1.c
NextPtr()	interact1.c
NodeNameComp()	string.c
OldStrCompStar()	string.c
Outputs()	commands1.c
ParseAliasFile()	parse.c
ParseCommands()	interact1.c
ParseOptions()	parse.c
ParseSimFile()	parse.c

PopFilePointer()	utils2.c
PopInt()	utils2.c
PopString()	string.c
PrintCode()	interact1.c
PrintHash()	utils1.c
PrintPaths()	commands3.c
PrintPrintFlags()	commands2.c
PrintWorkSpace()	commands3.c
PrintWSHeader()	commands2.c
Propagate()	commands4.c
PropagateClocks()	classgates2.c
PushFilePointer()	utils2.c
PushInt()	utils2.c
PushString()	string.c
PutPtr()	interact1.c
ReallocateWorkSpace()	interact1.c
Reset()	commands4.c
ResetWorkSpace()	interact2.c
ResolveNameList()	interact1.c
SetAlias()	commands4.c
SetDirection()	classgates1.c
SetPrintFlag()	commands2.c
SourceFile()	commands1.c
StrComp()	string.c
System()	commands2.c
ThresholdDrop()	commands1.c
ThresholdRise()	commands1.c

Biographical Note

The author was born May 22, 1951, in Seattle Washington. In 1958, he moved to Spokane, Washington and graduated from Mead High School in 1969. He entered Lewis and Clark College and received a Bachelor of Science degree in Chemistry in 1974.

In fall of 1981 he enrolled full time in classes at Portland State University, taking a mixture of computer science, engineering and mathematics classes. In fall of 1982 he began his studies at the Oregon Graduate Center.

During the summer of 1983, the author accepted a temporary position as an Intern Engineer/Scientist at the Computer Research Lab at Tektronix. In the fall of 1983 he accepted a permanent part-time position at Tektronix while still attending classes part-time at the Oregon Graduate Center. In the spring of 1984 when all class work for his Master of Science degree at the Oregon Graduate Center was completed, the author's status at Tektronix was changed to full-time.